Behrooz File System (BFS)

by

Behrooz Shafiee Sarjaz

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2015

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In this thesis, the Behrooz File System (BFS) is presented, which provides an in-memory distributed file system. BFS is a simple design which combines the best of in-memory and remote file systems. BFS stores data in the main memory of commodity servers and provides a shared unified file system view over them. BFS utilizes backend storage to provide persistency and availability. Unlike most existing distributed in-memory storage systems, BFS supports a general purpose POSIX-like file interface. BFS is built by grouping multiple servers' memory together; therefore, if applications and BFS servers are co-located, BFS is a highly efficient design because this architecture minimizes inter-node communication. This pattern is common in distributed computing environments and data analytics applications. A set of microbenchmarks and SPEC SFS 2014 benchmark are used to evaluate different aspects of BFS, such as throughput, reliability, and scalability. The evaluation results indicate the simple design of BFS is successful in delivering the expected performance, while certain workloads reveal limitations of BFS in handling a large number of files. Addressing these limitations, as well as other potential improvements, are considered as future work.

Acknowledgements

I would like to thank my supervisor, Professor Martin Karsten, for his help and support throughout my studies at the University of Waterloo. Further, I thank my thesis readers, Professors Bernard Wong and Ken Salem, for their insightful comments.

Dedication

I dedicate my thesis work to my family. A special feeling of gratitude to my loving parents whose words of encouragement and support never let me feel alone along this path.

Table of Contents

List of Tables ix

List of Figures x

1 Introduction 1

2 Design 4

2.1 Terminology . . . 4
2.2 System Overview . . . 4
2.3 BFS Namespace . . . 5
2.4 BFS Internals . . . 8
2.4.1 BFS File interface . . . 8
2.4.2 Data Model . . . 10
2.4.3 FUSE Library . . . 12
2.4.4 BFS Implementation . . . 14
2.4.5 Load Balancing . . . 16
2.4.6 Leader Node . . . 17
2.4.7 Membership and Failure Detection . . . 18
2.5 Backend Storage . . . 18
2.6 CAP Properties . . . 19
2.7 Consistency . . . 20

2.7.1 Consistency in BFS . . . 21
2.7.2 GlusterFS Consistency Model . . . 22
2.7.3 Openstack Swift Consistency Model . . . 22
2.8 BFS Servers Interconnect . . . 23
2.9 Access Control in BFS . . . 25

3 Related Work **26**

4 Evaluation **34**
4.1 Testbed Environment . . . 36
4.2 Throughput . . . 37
4.3 Reliability . . . 40
4.3.1 *fsync* Latency Test . . . 41
4.3.2 Crash Recovery Evaluation . . . 41
4.4 Scalability . . . 44
4.4.1 Usage Scenarios . . . 45
4.4.2 SPEC Workloads . . . 45
4.5 BFS_ZERO . . . 48
4.5.1 Latency comparison . . . 48
4.5.2 Throughput Comparison . . . 49

5 Future Work **51**

6 Conclusion **53**

APPENDICES **55**

A ZooKeeper Architecture **56**

B Testbed Configuration **58**

C SPEC SFS 2014 File Operations 59

D SPEC SFS 2014 Configuration File 60

References 61

List of Tables

3.1 Summary of different storage systems 33

4.1 Benchmarks Summary. • indicates the benchmark can be used for evaluating the corresponding file system dimension; ◦ is the same but the benchmark does not isolate a corresponding dimension; ⋆ is used for traces and production workloads. Taken directly from [42] 35

C.1 SPEC SFS 2014 File Operations, Taken directly from [18] 59

List of Figures

2.1 BFS Architecture 6
2.2 BFS Namespace using ZooKeeper 9
2.3 Granularity in BFS interaction with different sub-systems 12
2.4 VFS as a middle layer between file systems and processes 13
2.5 FUSE Architecture 13
2.6 BFS internal architecture and interaction with external systems 15
2.7 Sequential Consistency in BFS 21
2.8 Sequential Consistency in Swift 23
2.9 PF_RING Architecture 24

4.1 Read throughput evaluation 38
4.2 Write throughput evaluation 40
4.3 *fsync* Latency Test 42
4.4 Recovery Time in BFS with a varying number of crashed servers 43
4.5 SPEC SFS workloads 47
4.6 Latency for 4 KB blocks in microseconds 50

A.1 ZooKeeper Architecture 57

Chapter 1

Introduction

A diskless node is a workstation that provides no local hard disk and employs remote file systems for booting and storage. Using diskless nodes is not a new trend and is utilized in several environments. For example, in High Performance Computing (HPC), diskless nodes (thin clients) are used to provide computing, while storage requirements are served by a remote file system. In addition, in many large organizations, workstations are used in a similar fashion, where data and applications are stored remotely and are accessed locally. Further, in cloud computing using diskless nodes is highly desirable because removing disks reduces power consumption, space, heat, and noise [37]. In fact, separating storage concerns from processing concerns has become a predominant trend in most large-scale applications. For example, it is a common pattern for many applications to store their data on separate cloud storage systems and focus on the computing.

The origin of Behrooz File System (BFS) goes back to a diskless environment in an experimental operating system kernel, KOS [12], for many-core computer systems. Simplicity is one of the main design goals in KOS. One way of achieving it is to have no entrenched dependency on local hard disk; on the other hand, a POSIX-compliant file system is required to bootstrap and satisfy applications' storage needs. Therefore, the choices are similar to those of diskless node environments including using in-memory or remote file systems. Using simple in-memory file systems such as Linux tmpfs [39] is reasonable for bootstrapping the kernel; however, because of lack of a persistency, they cannot fulfill storage requirements of applications. Further, although remote file systems such as GlusterFS [6] or Ceph [46] are attractive options, they are fairly complicated systems depending on a non-trivial software stack. Even a simple NFS client [38] has various software dependencies. Further, using NFS limits KOS to only NFS server while there are a broad range of storage systems with superior performance compared to NFS. As a result, the

BFS project was initiated to combine the best of in-memory and remote file systems. BFS stores files in main memory, and it uses a persistent network-based backend storage for persistency. Originally, BFS was conceived to fulfill the storage requirements of an individual workstation; however, later the project was expanded to group the main memory of a collection of workstations to create a large scale storage system backed by a persistent storage.

BFS' goal is to completely eliminate the need for disk storage while providing persistent, low latency, and high throughput file I/O using main memory and network-based backend storage. In addition, providing a POSIX-compliant interface and feasibility for system bootstrap are important design goals for BFS. Finally, BFS tries to address these challenges by a simple, straightforward design and comparable performance to other (more complex) solutions.

In this thesis, BFS is presented as an endeavour to address the aforementioned goals. In BFS, data is stored in the main memory of commodity servers. In fact, the main memory of many servers is aggregated to create a large-scale storage system. In order to provide persistency, BFS uses backend storage. Backend storage is any type of storage system that provides persistency guarantees. A backend storage can be a traditional file system that provides strong consistency, or a modern distributed file system that provides high availability and scalability. Data is not replicated in BFS due to the scarcity of main memory; instead, replication is provided by the backend storage system of choice. BFS scales by adding more nodes with memory and expanding the backend storage. Therefore, BFS can be built and used by a single node or by a cluster of several nodes. BFS uses main memory to provide low-latency and high throughput I/O; hence, the intercommunication between BFS nodes should be fast enough to not significantly increase the latency or decrease the throughput of the main memory. As a result, BFS is efficient in environments such as data centers, where fast and low-latency networking is available.

BFS is built by grouping multiple nodes' memory together and providing a shared unified file system view over them. If applications and BFS servers are co-located, BFS would be a highly efficient design because this pattern minimizes inter-node communication. In fact, in this case BFS performs similar to an in-memory file system since files are mainly served from the main memory of machines on which applications are running on. This pattern is very common in distributed computing environments, where each node performs some computation and produces some data that need to be stored and shared with other nodes. Examples of such environments include HPC or data analytics, such as Hadoop-based applications [8]. On the other hand, when applications and BFS servers are not co-located, BFS is similar to a remote file system. Remote file systems are limited by the network and remote storage. However, BFS is limited by network and remote nodes' mem-

ory and for read operations, and for write operations it is limited by network and remote storage. Therefore, it is expected for BFS to have a comparable and superior performance than remote file systems when applications and BFS servers are not co-located. Evaluation results of BFS using various workloads and benchmarks indicate that the simple design of BFS is successful in delivering the expected performance. However, certain workloads such as those with a large number of files reveal important shortcomings in BFS that are addressed as future work. BFS is publicly available at http://bshafiee.github.io/BFS/.

Chapter 2

Design

BFS is built by creating a global namespace over the main memory of commodity servers. BFS uses ZooKeeper [25] to create a global consistent namespace and utilizes different backend storage systems to permanently store files. BFS provides an interface similar to the POSIX standard [16] using the FUSE library [5]. BFS employs several techniques to create a low-overhead communication mechanism between nodes. In the following, an overview of BFS is presented, followed by a detailed explanation of each part of the system.

2.1 Terminology

In this thesis, *BFS server* or *server* refers to any machine that BFS uses to store files. *BFS application* or *application* refers to any program that mounts and uses BFS. *BFS client* or *client* refers to any machine that runs a BFS application but is not a BFS server. BFS applications can run on BFS clients as well as BFS servers. Note that a BFS client or server can run several applications that use BFS simultaneously. Finally, a *backend storage* is any storage system that is used by BFS to store files permanently.

2.2 System Overview

BFS aggregates the main memory of several machines (BFS servers) together and creates a global unified namespace among them. BFS uses the ZooKeeper consensus protocol to create this consistent namespace among participating servers. Different clients can use BFS

by mounting BFS to a directory just as any regular directory in any UNIX-like system. Clients can execute on BFS servers or any other machine. In fact, there is no distinct difference between clients and servers in BFS, except the fact that clients only modify files, while servers are used to store files as well. One of the BFS servers is known as the *Leader Node*. It acts as a regular BFS server with some additional responsibilities explained in Section 2.4.6.

Figure 2.1 shows a general overview of the BFS architecture. As it can be seen in this figure, all BFS servers use a global namespace to become aware of files that other servers are hosting. In addition, servers may communicate to move or manipulate each other's files. A BFS application can be running on any of BFS servers or separate machines, such as *C1* in Figure 2.1. Separate clients also use the global namespace to discover existing files. However, unlike servers, they only read from the global namespace and do not manipulate it because they do not participate in the BFS as storage servers. Finally, the last type of communication is the communication between BFS servers and the backend storage. Each BFS server is directly connected to the backend storage and uses it to permanently store files.

2.3 BFS Namespace

The BFS namespace is responsible to cohere BFS servers. Using this global namespace, servers become aware of other servers and their files. In addition, BFS uses this namespace for other tasks such as automatic node failure/membership detection (Section 2.4.7), leader election (section 2.4.6), and system statistics. In the following, first a brief review of different namespace architectures is presented. Then it is discussed how BFS creates its global namespace using ZooKeeper.

Naming plays a crucial role in computer systems. Names are used to uniquely identify resources in a computer system and the process of identifying a resource is known as *Name Resolution*. Thus, a naming system is required to assign and resolve names. Naming systems organize names into a namespace. A namespace is a structure that holds a mapping of names to resources. For example, Domain Name Servers (DNSs) are responsible for resolving domain/host names to IP addresses in the Internet, and similarly in a file system, there is a naming service for resolving and assigning names to inode numbers.

There are two main namespace architectures, flat and hierarchical. In a flat architecture, names are symbols that are interpreted as a single unit, without any internal structure or relation. In contrast, in a hierarchical architecture, names are tied tightly to

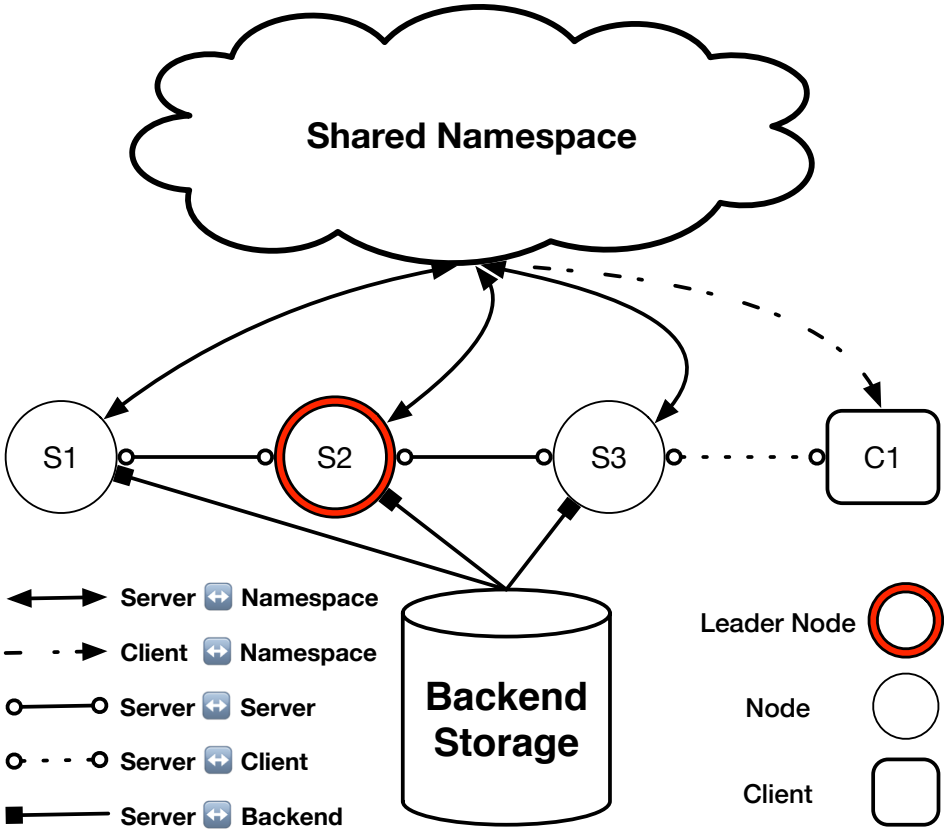


Figure 2.1: BFS Architecture

the hierarchy used for the names. It is the responsibility of a naming system to define the structure of the hierarchy, decide how the hierarchy is to be partitioned, and specify how names in the same partitions are related. Local file systems use a hierarchical namespace because they are designed to be human-friendly, while most object storage systems use a flat architecture because they provide an interface for other machines rather than humans. Similarly, most distributed file systems, which are designed to be used by humans, use a hierarchical namespace, such as GlusterFS [6], while distributed file systems such as Openstack Swift [14], which is usually used by other computer systems, use a flat hierarchy.

BFS provides a POSIX-like interface, which is designed for local file systems; thus, BFS uses a hierarchical namespace, although it might interact with object storage systems in the backend. For example, when Openstack Swift is used as the backend storage, BFS

emulates a hierarchical name space using a flat namespace. If a file in BFS has a full path of *"/Dir1/Dir2/File1"* it will be stored with the whole path as a single unique name. However, when GlusterFS is used as the backend, in order to store *"File1"*, first *"Dir1"* and *"Dir2"* need to be created if they do not exist.

Name resolution is an important part of a naming system. In local file systems, this functionality is straightforward because the file system stores all mappings of names to file inode numbers. Conversely, in a shared distributed environment, this is often not as straightforward. Some distributed file systems, such as early versions of the Google File System [24], use a central server as the name server; therefore, it is very similar to the local file systems approach. Although this design has the advantage of simplicity, it can suffer from drawbacks of having a single point of failure and the scalability concerns of a single central server. In contrast, some distributed storage systems such as Dynamo [22], Openstack Swift [14], or Ivy [34] use a symmetric approach in which each node in the system plays a role in providing the name service. Most of these systems use a DHT-based technique for lookups and distributing naming data among nodes. Ivy uses the Chord DHT system [40], whereas Dynamo, Openstack Swift, GlusterFS, and many other distributed file systems, use a DHT system known as Consistent Hashing [27] . DHT-based systems do not suffer from a single point of failure and are designed to be scalable; however, they are much more complex than a central design, and lookups are usually slower compared to a central design because each lookup request might go through several nodes.

BFS builds its naming system by choosing a hybrid approach. BFS uses a central shared server that is highly replicated. BFS creates this central shared server using the ZooKeeper library. ZooKeeper is a consensus protocol for coordinating processes of distributed applications. Details about ZooKeeper are presented in Appendix A. BFS utilizes the central namespace of ZooKeeper to store which files each BFS server is hosting. ZooKeeper significantly simplifies the name service design of BFS. ZooKeeper acts as a single name server in BFS, while this single server is highly replicated behind the scenes. Using ZooKeeper helps BFS achieving simplicity of a central server design without being concerned about having a single point of failure using replication. Each BFS server has a corresponding *directory* or *node* at ZooKeeper that holds the list of files this BFS server is hosting. Figure 2.2 depicts how BFS builds its namespace over the ZooKeeper namespace. When a BFS server creates a new file, it updates the corresponding directory in the ZooKeeper namespace, and this change is reflected using ZooKeeper *watches* to all other BFS servers. ZooKeeper watches are a cheap mechanism to receive timely notifications of changes without requiring polling. Upon receiving a change notification, a BFS server reads the name data of a changed directory at ZooKeeper and updates its own namespace information. In addition, when a BFS server crashes, a watch event is sent to every other server, and all BFS servers realize

a server has crashed and remove the missing files from their mapping. Similarly, when a BFS server joins ZooKeeper, a watch event is triggered to all other servers to inform them about the newly joined server. In this way, each server in BFS has a global view of all files and servers in the system.

Using watches for propagating changes can introduce a consistency problem when two applications running on two different BFS servers have their own communication channel. For instance, consider application A on the first server creates a new file and informs application B on the second BFS server through their communication channel. Accordingly, application B is aware of the newly created file on A while B might not have received the change notification through ZooKeeper watches yet. This is a violation of the strong consistency semantics of a local UNIX file system; however, as it is explained in Section 2.7.1, BFS cannot guarantee strong consistency when it is used with a non-consistent backend storage anyway.

Although using ZooKeeper for building a shared name space in BFS significantly simplifies the design and implementation of BFS; it can become a bottleneck under certain workloads. For example, consider a scenario in which a single BFS server is creating and storing a large number of small files. Then, the size of the corresponding directory significantly increases, which is costly for ZooKeeper to store and replicate. ZooKeeper is designed to handle small data nodes. In addition, this problem intensifies when the number of BFS servers with a large number of small files increases; this effect is studied in Section 4.4, and results indicate that ZooKeeper becomes very slow under many-file creation/deletion workloads. Possible solutions to alleviate this situation are discussed in Section 5.

2.4 BFS Internals

In this section it is discussed what interface BFS provides, how BFS stores files, what granularity BFS uses to store files, and what BFS does in case of failure or new server membership.

2.4.1 BFS File interface

BFS uses the FUSE library to provide a traditional POSIX-like file interface. The term POSIX-like is used instead of POSIX-compliant because when BFS is used with a non-consistent backend storage, BFS cannot provide strong consistency guarantees of the

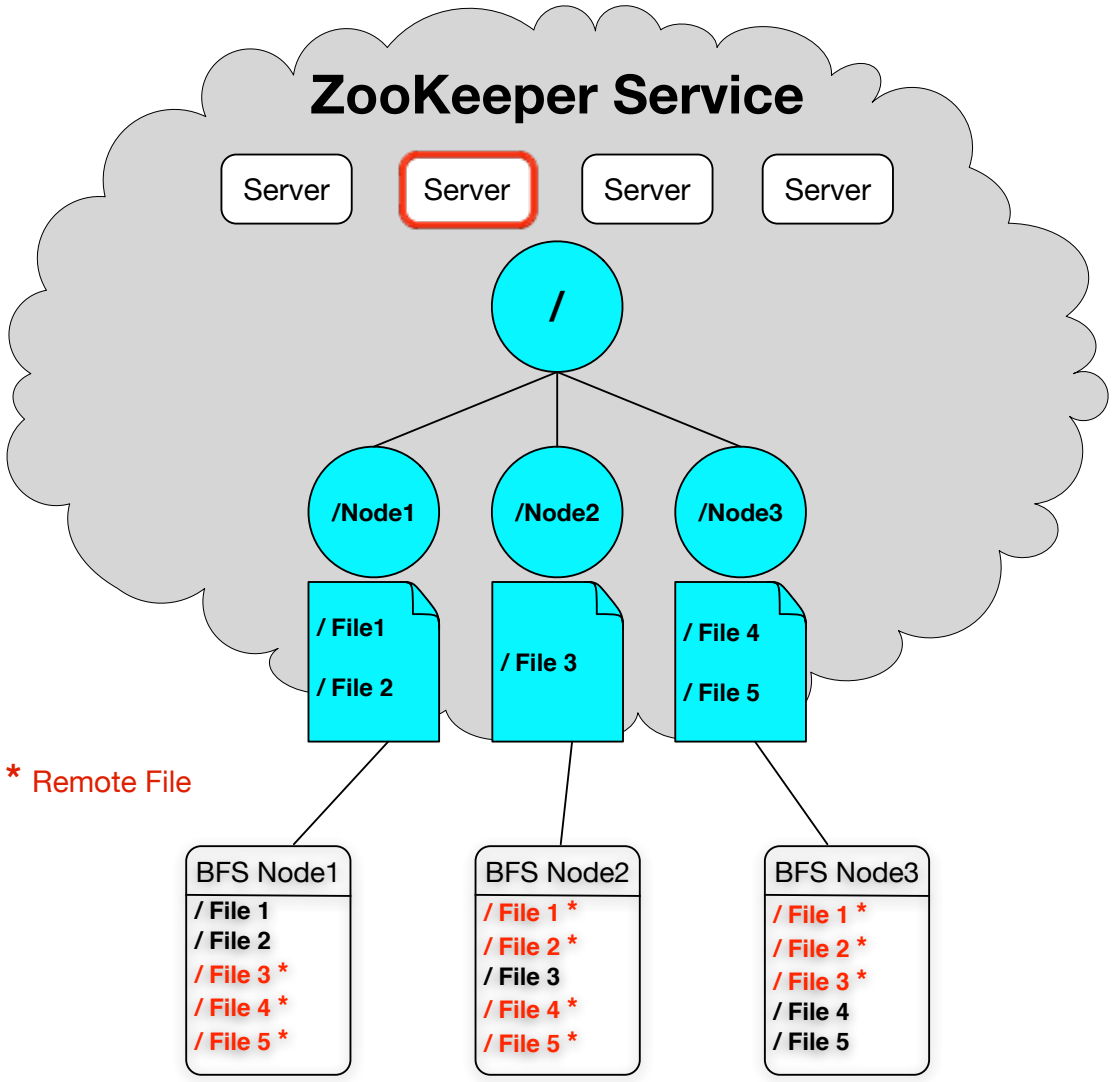


Figure 2.2: BFS Namespace using ZooKeeper

POSIX standard. However, as explained in Section 2.7.1, using a consistent backend storage ensures BFS a POSIX-compliant system. On any BFS server or client, BFS can be mounted under any regular directory, and applications can create, delete, read, or manipulate files just as they were stored on a local disk. Furthermore, an application that uses BFS is not able to distinguish whether the files it is accessing are accessed locally or remotely. Detailed explanation of BFS consistency guarantees are provided in section 2.7.1. In addition, some features that usually exist in common POSIX-compliant file systems, are not available in BFS. Examples of such features includes extended attributes, POSIX file locking, symbolic links, and hard links.

2.4.2 Data Model

In order to store or retrieve data to/from a file system, a well-defined data model is required. A data model defines the basic nature of entities stored on a medium. Storage entities can be structured or unstructured. For example, a byte, a group of bytes known as a block, or a group of blocks knows as an object are examples of unstructured entities, whereas a table of a relational database is a structured storage entity. Structured data models offer convenience and more power to perform data extraction; however, they incur a high overhead and slowdown upon a storage system compared to unstructured data models.

Data granularity in a data model specifies how big storage entities are. It has been investigated by different systems with no universal solution. For instance, using a small granularity increases the size of metadata, while using big granularity can waste storage and cause fragmentation. Traditional file systems usually choose a block as the storage granularity, but they expose a byte-level access to the files. This approach has been successful for many years and is suitable for serving a moderate number of files. On the contrary, cloud computing workloads usually process a massive number of files, which can be several orders of magnitude larger than everyday applications' file size. Traditional file systems can not scale with such workload characteristics due to the contention in the file system and massive amount of metadata generated for these huge workloads. To serve such environments, object level storage are proposed that use a bigger data granularity, an object (a group of blocks). Despite all the benefits of object storage systems, they are not a suitable choice for use cases where there are a lot of small I/O operations and demand for operating at byte-level. Although some object storage systems such SGI OmniStor [17] and those that are compatible with the Cloud Data Management Interface (CDMI) [2] have support for byte-level operations, most commercial and widely used object storage systems, such as Amazon S3 [1], Google cloud storage [7], or Openstack Swift [14], do not support byte-level operations.

In conclusion, there is no universal best data model for a file storage system. As discussed in the previous section, each choice has different pros and cons. Moreover, considering the simplicity of the exposed interface and interoperability with other systems in a computer, such as main memory or networking, adds to the complexity of choosing a data model for a file storage system.

BFS Data Model

BFS interacts with different storage sub-systems of a computer at the same time. BFS uses memory as the local storage, stores files to a backend storage (block-level or object storage), and transfers data between nodes using a network connection. In addition, BFS exposes a POSIX-like interface to applications. Thus, choosing a data model in BFS is different from regular storage systems.

Figure 2.3 shows the data granularity in BFS interactions with different sub-systems. BFS interacts with applications using a POSIX-like interface, which operates on byte-level. Similarly, a BFS server interacts with other BFS servers over the network using the regular network interface, which is used by a byte-level interface. BFS utilizes an internal block size to store files in the main memory. Finally, BFS stores data permanently at the backend storage, which can be a block-level storage such as GlusterFS or an object storage system such as Openstack Swift.

BFS chooses a block as the granularity for storing files in memory, and when BFS uses an object storage as the backend, it maps each file to an object. BFS exposes a byte-level interface, which allows partial reads/writes. On the other hand, most object storage systems, including Openstack Swift, which BFS uses as a backend storage, do not support byte-level operations. To handle this situation, BFS performs byte-level operations on existing files in the memory and updates the whole object in the backend in case of an update. The main drawback of this approach is that uploading the whole file instead of affected bytes (in case of modification) to the backend can take significantly longer than only uploading modified bytes to the backend. One solution to this problem is to shrink a file to multiple objects that are fixed-size but are bigger than blocks. The main reason for using blocks as the data granularity for storing files in BFS is simplicity, but as explained before, the proper size of a block needs to be studied further. In fact, the choice of granularity affects many design decisions for BFS, such as interactions with the backend storage, handling overflows (refer to Section 2.4.5), the exposed user interface, efficiency, and performance of the system. Thus, this topic needs to be studied further and is considered as future work.

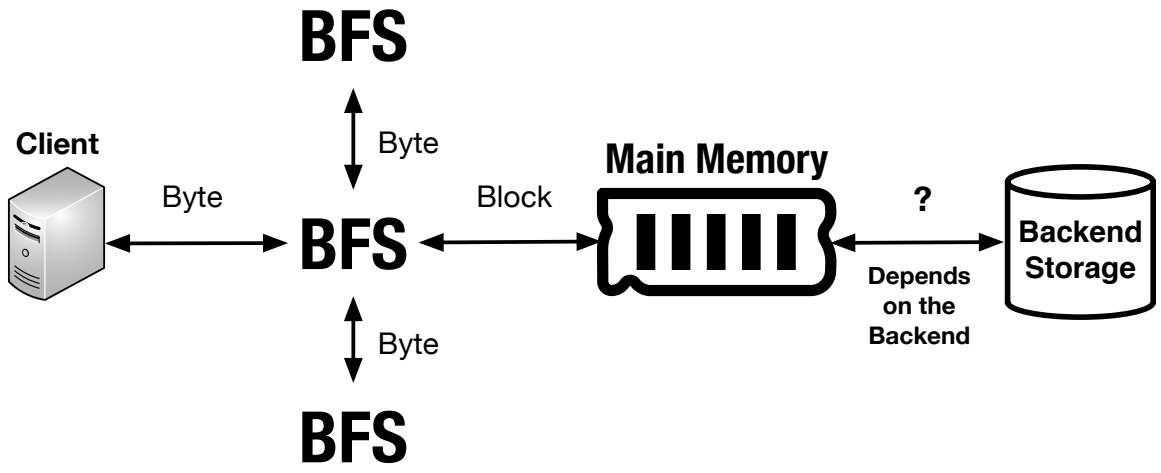


Figure 2.3: Granularity in BFS interaction with different sub-systems

2.4.3 FUSE Library

BFS uses the FUSE library to provide a POSIX-like interface to applications. File systems are normally implemented as a part of the operating system kernel. FUSE provides a mechanism to implement a file system without having to change, compile, or know about the kernel. Most UNIX-like operating systems use a layer called Virtual File System (VFS) between file systems and the kernel. The purpose of VFS is to access different file systems using the same interface. For example, using VFS, Linux users can have the same and indistinguishable access to different file systems, such as ext3, ext4, NTFS, or any other type of supported file system. In addition, VFS allows updating a file system without having to change or update the kernel. In fact, VFS resides between different user space applications and different file systems as depicted in Figure 2.4.

FUSE also utilizes the VFS facility to provide a file system in user space. FUSE has two main parts, a kernel module and a user space library called libfuse. libfuse exports the FUSE API to file system developers in user space, and the kernel module interacts with VFS and forwards requests and responses to/from libfuse using a character device. A general overview of the FUSE architecture is presented in Figure 2.5.

In this diagram, a user space file system is registered in FUSE using libfuse. The user space file system is mounted under *"/mountdir"* directory, and this file system is accessed by *Process 1* through that directory. All file operation on *"/mountdir"* are captured by

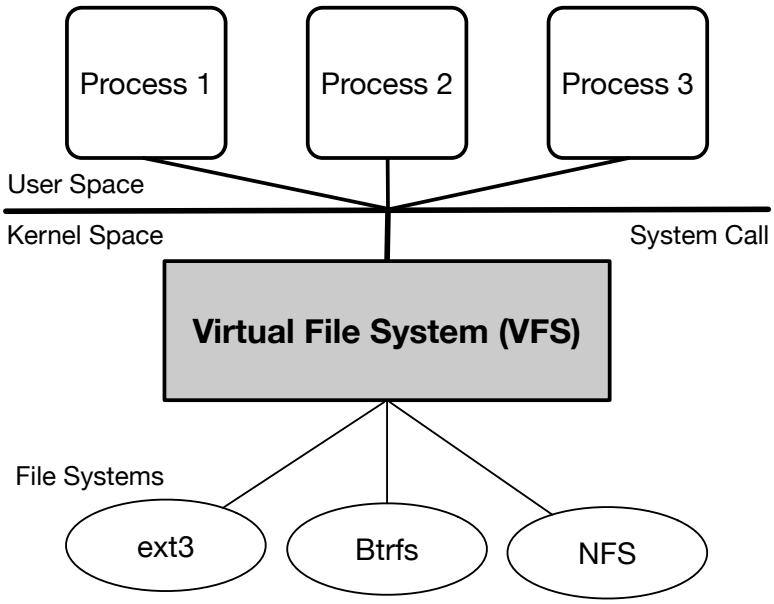


Figure 2.4: VFS as a middle layer between file systems and processes

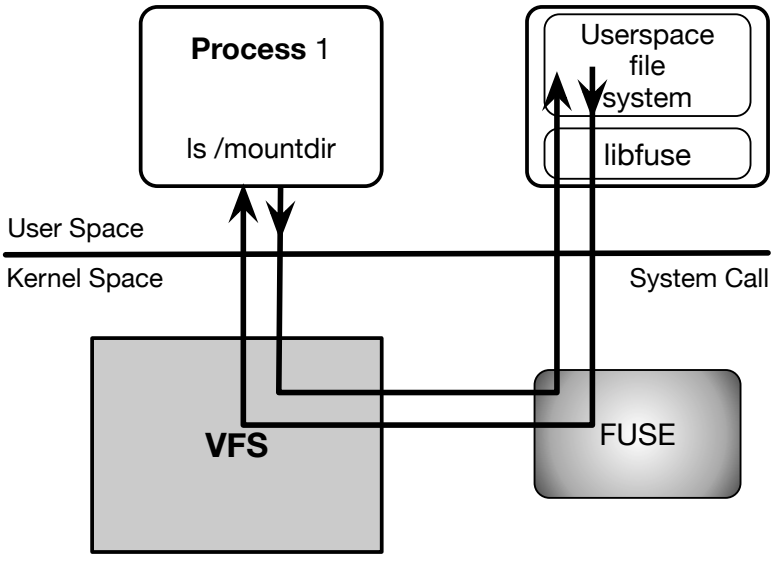


Figure 2.5: FUSE Architecture

VFS and then delivered to the FUSE kernel module. In the next step, the FUSE kernel module forwards file operations to libfuse and finally to the user space file system process. For instance, when *Process 1* runs the *ls* command (*readdir*) against *"/mountdir"* the request is captured by a system call to VFS, routed to FUSE kernel module, then to libfuse, and finally to the registered user space file system. Upon receiving the *readdir* request, the user space file system processes the request and sends back the reply through the same channel. BFS has been implemented as a user space file system by means of the FUSE library and captures and processes' file I/O requests as described.

2.4.4 BFS Implementation

Once a file I/O request is delivered to BFS via FUSE, the BFS storage engine processes the request and serves it if the required file is stored locally, or forwards the request to the remote server where the requested file is stored. BFS has been implemented in C++ in approximately 12 K lines of code. Figure 2.6 shows a general overview of BFS internal architecture.

BFS has six main internal sub-systems. File I/O requests from libfuse are received and pre-processed by the FUSE Callback Handler. In the next step, the request is handed to the In-Memory Storage Engine that is the central part of the system. If the requested file is present on the running server, it will be served locally, and results are written back through the FUSE Handler. Necessary synchronization actions are taken by the Sync Engine that communicates with the backend storage. For instance, if a write has happened to a file, the Storage Engine asks the Sync Engine to reflect the changes in the backend storage. Furthermore, it is important to mention that write request are not blocked for synchronization with the backend storage, unless a synchronized I/O operation is requested. For instance, if flush, close, or fsync are called or if the file has been opened with the **O_SYNC** flag all modification to the file are synchronized with the backend storage before returning from the FUSE call. Consequently, I/O operations are performed in an asynchronous manner unless explicitly synchronized I/O is requested.

When a request is received for a file that is not present on the current server, Storage Engine determines which server is hosting the requested file using the Namespace Handler, and it sends a request through the Interconnect Handler to that remote server. It is worthwhile to mention that Namespace Handler caches the mapping and does not send a query to ZooKeeper each time to look up the host server for a file. Finally, once the result of a remote request is received by the Interconnect Handler, it hands back the result to the FUSE Handler and finally back to the application. In addition to the five mentioned

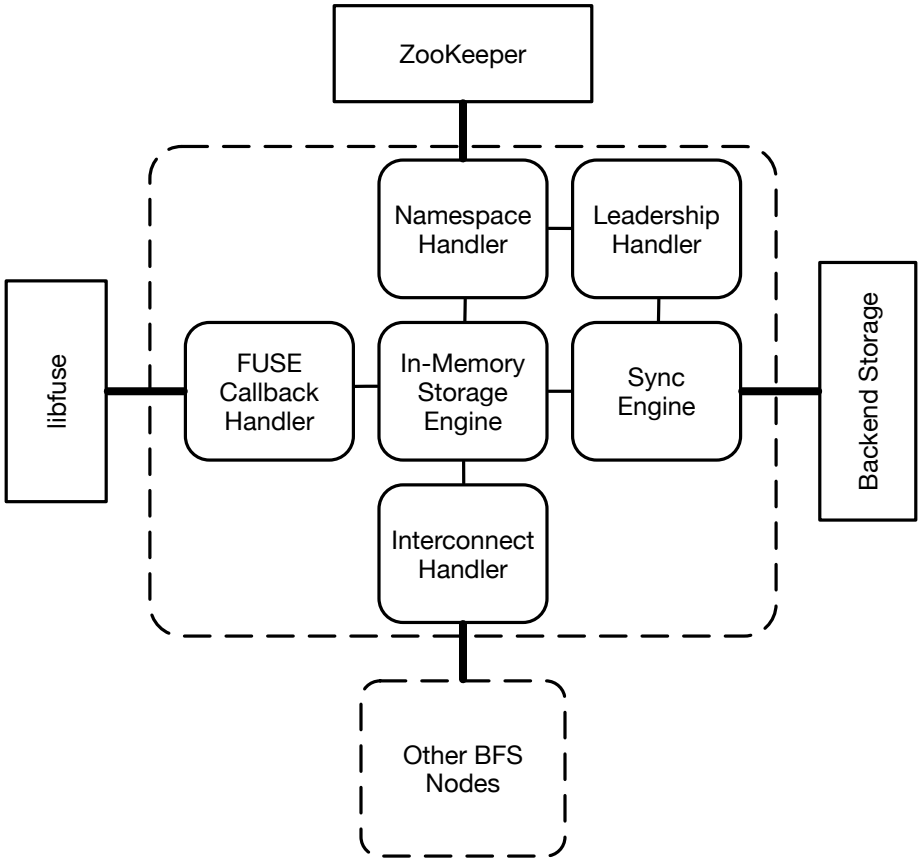


Figure 2.6: BFS internal architecture and interaction with external systems

subsystems in BFS, there is a Leadership Handler module that is discussed in detail in Section 2.4.6.

2.4.5 Load Balancing

There are two different scenarios in which BFS needs to perform load balancing. First, BFS can proactively move files from a server that is under heavy load to other servers. For example, in some distributed applications only a single server is producing data, and the rest mostly use data. Therefore, the BFS server that hosts the producer, will experience significant load and contention from readers, whereas those BFS servers that host consumer applications do not experience any storage load. Therefore, it is beneficial to proactively split load among different BFS servers. In addition, consider a client creates a file on server A, and another client repeatedly accesses that file from server B. In this case it would be beneficial to move that file to server B; as a result, most accesses to the file will happen locally. At this point, no proactive load balancing mechanism is implemented in BFS; however, as explained in Section 3, this is one of the main topics to be considered for the future road map of BFS.

Another scenario for load balancing is to actively move files from a BFS server that has reached its maximum capacity to other BFS servers. By default BFS writes all the written data to the local server that hosts the file; hence, when a server reaches its maximum capacity, BFS needs to find other free servers to continue writing. There are several options to handle this situation. First, the file that is being written can be moved to another server with enough free space. Second, some existing files can be moved to other BFS servers to make free space for the file that is being written. Third, the part of file that is already hosted on the current server can be kept, and the rest of file can be written on another free BFS server.

In the first approach, BFS moves the file that is being written to another server. This approach is implemented in BFS and is used as the default moving policy. Although it seems this approach is not considering the locality of file accesses, it has the advantage of simplicity. It is easy to implement this approach because it affects the least number of files in the system (only moves the current file to another server).

The second approach is inspired by the eviction techniques in cache, where least recently used files are considered as good candidates for eviction. This approach keeps the locality of file accesses by moving the files that are accessed the least. However, an important question is how much free space is required to accommodate the newly written data. For instance, consider the newly written buffer size is 1 MB, and the host server has reached its

maximum capacity; therefore, at least 1 MB free space should be made available. If only 1 MB new free space is created, the next write to this file, which is very likely to happen, will experience the same issue. In the world of programming languages, resizable arrays also experience this issue. Most programming languages use an exponential approach and double the size of the array each time maximum capacity is reached. The same semantics can be used in determining how much free space is required for expanding files as well. However, another problem with this method happens when there are many small files or very few massive files. For instance, assume that a BFS server with 20 GB capacity has become full, and the current file being written is 1 GB. Further, assume this server contains another 19 GB file. Therefore, the least recently used file is the second 19 GB file and moving it to another node to make space for 2 GB is considerably inefficient. Similarly, assume the same BFS server but with thousands of tiny files instead of a single 19 GB file. As a result, making 2 GB free space requires to move many of these tiny files that is inefficient compared to moving a single file of the same size (first approach). Due to the aforementioned issues, this policy has not been implemented in BFS.

In the third approach, only the newly written data is moved to another BFS server, and existing chunks of the file (as well as other files the server is hosting) are kept at the same BFS server. In addition, it only moves the newly written data to other servers; thus, it has the minimum cost of moving compared to the last two approaches. On the other hand, as explained in Section 2.4.2; this mechanism requires another level of metadata for keeping track of where each part of a file is hosted. This significantly increases the complexity of implementation and namespace design. Hence, this technique is not implemented in BFS.

In conclusion, none of the above approaches are perfect, and each has its own pros and cons. A comprehensive solution might be to use a hybrid approach that could alternate between different policies and decide based on the situation. As explained in Section 5, this issue is one of the topics to be considered as future work for BFS.

2.4.6 Leader Node

BFS uses a distributed leader election algorithm by means of ZooKeeper to choose one of the BFS servers as the leader node. Details of leader election algorithm are provided in [25]. The leader node is same as any other BFS server with two additional responsibilities that are described in the following paragraphs.

First, the leader node is responsible for handling other BFS servers' crashes. When a BFS server crashes, the leader uses the global namespace to determine which files are missing (files that exist in the backend storage but are not present in any BFS server), then

it asks a subset of BFS servers to fetch missing files from the backend storage. If the leader node crashes, another BFS server is chosen as the leader via ZooKeeper. Consequently, as long as there is one or more servers present, there will be always a leader among BFS servers.

Second, having a leader is important during the bootstrap process with an existing backend storage. For instance, consider a situation where BFS is going to run with an existing backend storage that contains many files. Similar to the failure scenario, the leader asks different BFS servers to fetch missing files from the backend storage. It is worthwhile to mention that the leader in both scenarios (failure and bootstrap) assigns missing files to BFS servers according to their free storage capacity.

2.4.7 Membership and Failure Detection

A crucial task in any distributed systems is to detect and handle new server membership and failures. As described in the previous section, BFS handles server failures using the leader node. However, the first issue to address is to detect failures. BFS uses the group membership facility of ZooKeeper to detect join and departure of servers. As explained in Section 2.3, each BFS server upon arrival creates a directory in the ZooKeeper namespace, and ZooKeeper informs existing BFS servers about the arrival of new servers using so-called *watches*. BFS servers create so-called *Ephemeral* nodes in ZooKeeper. An ephemeral node in ZooKeeper is a directory that will be either deleted explicitly by the system or is automatically deleted when the session between ZooKeeper and BFS server that created the ZooKeeper node expires. Once an ephemeral node is deleted other BFS servers are informed through a watch event. Session expiration might be due to server failure, networking issues, power outage or any other reason that disconnects a BFS server from ZooKeeper. More details about how ZooKeeper discovers session departures is provided in the original paper [25].

2.5 Backend Storage

Backend storage is an important part of BFS design. It is responsible to permanently store data; in addition, BFS can take advantage of several other features that a backend storage solution might provide. For instance, BFS does not replicate files for availability because if replication is required, it can be provided using a replicated backend storage. There are several distributed object storage solutions that provide a high level of availability through replication, such as Openstack Swift.

BFS is designed to be compatible with different types of backend storage. In fact, BFS is be able to work with any storage solution. BFS uses a middle layer between its storage engine and backend storage. Therefore, adding support for a new storage solution is only the matter of implementing the virtual interface of this middle layer. This middle layer handles how each file in BFS is mapped to a storage entity in the backend. For instance, the current OpenStack Swift backend plugin maps each file to an object in the Swift object storage, while the GlusterFS plugin has a file to file mapping. This design simplifies BFS communication with different backend storage and allows to extend backend storage support easily. At the time of writing this document, an object-based storage solution, Openstack Swift, and a block-level storage, GlusterFS are supported in the BFS as backend storage, and it is easy to add support for other storage solutions as well.

2.6 CAP Properties

The CAP theorem [21] is a widely accepted framework to describe issues in any distributed system, and how these issues affect each other. Any distributed system's goal is to provide all of the following properties at the best level:

Consistency
: A total order must exist on all operations as if the operations were completed at a single instance. For example, in context of a distributed file system, any subsequent read after a completed write operation must return the value of this (or a more recent) write operation.

Availability
: Each request received by an operating node should receive a response in a described time frame.

Partition Tolerance
: The system is working properly in case of arbitrarily losing many messages sent from one node to another.

The CAP theorem states that although all of these properties are desirable, a distributed system can perform well in at most two of these properties simultaneously. As a result, one of these properties should be relaxed to achieve the other two. For instance, if consistency promises are relaxed, the system can always reply to requests even if nodes cannot communicate; however, there is no guarantee that the returned data is consistent.

BFS can not be directly categorized in terms of the CAP theorem because unlike typical storage systems, BFS has two different layers of storage. First, the frontend where files are stored and accessed in the main memory of BFS servers. Second, the backend storage where files are stored permanently. The BFS frontend only stores a single replica of each file; thus, it provides strong consistency. In contrast, the lack of replication makes BFS frontend unavailable in case of server failure or generally network partitioning. On the other hand, files are stored in the backend storage as well. Therefore, if a highly available backend storage (e.g. Openstack Swift) is used, BFS provides high availability. However, AP (Available and Partition-Tolerant) storage sacrifice consistency; as a result, strong consistency promises of BFS frontend are also relaxed in favor of availability. Further, if a consistent backend storage such as GlusterFS is used, then BFS will provide consistency with a weaker availability in the case of network partitioning. In conclusion, BFS consistency and availability promises depend on the backend storage that is used. Due to the importance of consistency in storage systems, a more thorough discussion of consistency in BFS in different scenarios is presented in the next section.

2.7 Consistency

A consistency model is an agreement between a data store (e.g. a file system) and clients that use the data store [41]. Normally, a process expects to see the result of the last write upon a read operation on a file. However, in a distributed system due to the lack of a global synchronized clock, it is difficult to determine what is considered the last write. Moreover, due to the possibility of network partitioning, a node might not be aware of writes happening at other nodes. Therefore, in such systems alternative definitions are required that essentially narrow the valid results of a read operation. Each of these definitions are known as a consistency model.

Two important consistency models are *sequential* and *eventual* consistency models. In the sequential consistency model, it is expected that operations on a data store be executed in the order specified by applications [41]. In other words, this definition means that any interleaving of operations is valid as long as all applications see the same interleaving of operations. Eventual consistency is a very simple and relaxed consistency model in which all replicas of a data item will become the same, if no update happens for a long period of time [41]. Eventual consistency model usually describes the consistency promises of AP distributed systems.

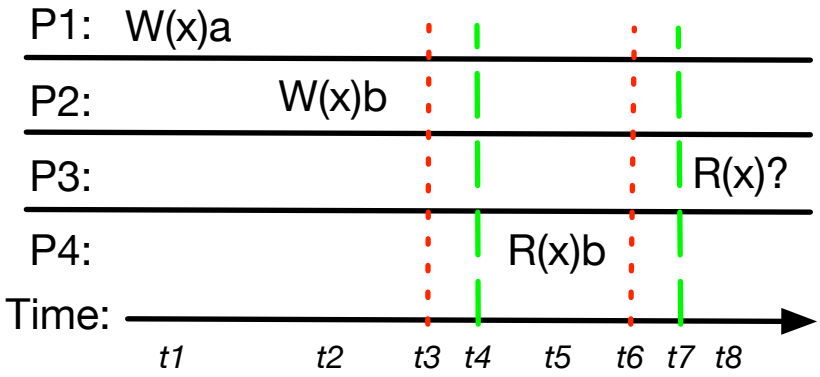


Figure 2.7: Sequential Consistency in BFS

2.7.1 Consistency in BFS

The BFS consistency model is tied to the backend storage. However, more specifically, BFS consistency guarantees depend on the backend storage, if and only if a node failure happens. Normally, all operations are performed on a single server's memory, but when a server crashes or a network partitioning happens, there will be a temporary unavailability of missing files until they are recovered from the backend storage. After recovery, the content of recovered files depends on the consistency model of the backend storage. For instance, if a consistent backend storage is used, BFS still provides strong consistency guarantees. Therefore, during a temporary unavailability, BFS does not violate consistency guarantees and only blocks the requests until the missing files are fully recovered from the backend storage, which means a weaker form of availability. In fact, if BFS is seen as a distributed cache over a data store, it is strongly consistent in normal scenarios (without a crash), which is not the case for CPU cache or file cache; furthermore, after a crash, BFS provides a weaker availability and the content of recovered files depends on the backend storage.

BFS is sequentially consistent as long as the underlying backend storage is sequentially consistent. For instance, if the backend storage is not sequentially consistent the client might see sequentially inconsistent behaviour as well. Consider Figure 2.7, *Process 1* writes the value of a on file x at time *t1* and receives acknowledgement from BFS about completion of the write (flushed write) operation, denoted as *W(x)a* by *P1* at *t1*, and *Process 2* successfully writes b to x at time *t2*.

At *t3* the primary server for x crashes, and the system experiences a temporal unavail-

ability until *t4* for file *x*. At *t5*, *P4* reads the recently recovered *x* and receives *b* as the value of *x*. At *t6*, the underlying node for *x* crashes again, and the system suffers from temporal unavailability until *t7*. Finally at *t8*, *P3* reads *x* from the underlying server for *x* which has finished recovering *x* from the backend at *t7*. If BFS backend storage is a sequentially consistent data store, BFS will return *b* as the result of a read, and in the case of a non sequentially consistent backend storage, BFS might return *b*, *a* or any write value before *a*.

Finally, it is important to remember that the described consistency model for BFS applies to file modification operations (e.g read or write). As described in Section 2.3, file creation/deletion are propagated to BFS servers via ZooKeeper *watches*. ZooKeeper delivers *watches* in an asynchronous manner, and this leads to an eventual consistency model for file creations/deletions.

2.7.2 GlusterFS Consistency Model

GlusterFS is POSIX-compliant, which means it provides a strong consistency model. GlusterFS provides strong consistency at the cost of relaxing availability. In other words, if a network partitioning happens then the partitioned nodes may not have consistent replicas; therefore, GlusterFS is not able to provide a consistent response. Hence, in order to preserve consistency, it does not provide a response and availability is sacrificed. On the other hand, newer versions of GlusterFS have support for options that loosen strong consistency to eventual consistency and preserve availability and partition tolerance.

2.7.3 Openstack Swift Consistency Model

Swift object storage provides eventual consistency, meaning that all replicas will become consistent, if no updates happens for a long period of time [20]. For example, if an object is written, Swift replicator starts to gradually update all replicas of that object to the latest version. Consequently, if a subsequent read happens on the mentioned object before the replication is finished, it might not necessarily see the latest version of that object. In terms of the CAP theorem, Swift sacrifices most consistency guarantees in favour of availability and partition tolerance. This means that even though consistency guarantees in Swift are highly relaxed, availability and partition tolerance are high.

Normally, Swift does not provide sequential consistency guarantees. Consider Figure 2.8, data item *x* is *a* at the beginning, then two writes are performed on *x* by *P2* and *P3*. The result of *R2(x)* can be any of *a*, *b*, or *c* values (Note that sequential consistency

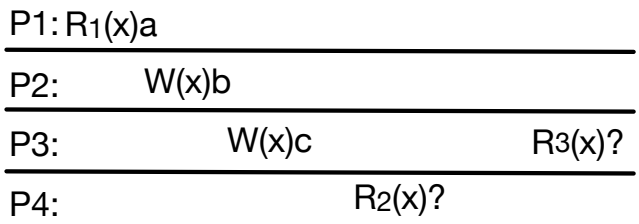


Figure 2.8: Sequential Consistency in Swift

has not been violated so far). Assuming *R2(x)* will return *b*, if Swift returns anything other than *b* or *c* for *R3(x)*, it will violate sequential consistency rules. In fact, swift might return *a* for *R3(x)* because by default it will return the value of first found replica of *x* without considering its version.

Replication in Swift is done using a Quorum-Based protocol [41], meaning that every read or write operation is served in agreement with a subset of replicas. For example, imagine each data item is replicated at N nodes in Swift, and agreement of R nodes for read operations, and W nodes from total of N nodes is required for write operations. Therefore, if the following condition holds, one can claim strong consistency for Swift [41]:

1. R+W >N (avoids read-write conflict)
2. W >N/2 (avoids write-write conflict)

As a result, Swift can be configured with relevant values for R and W to provide strong consistency. However, this will adversely affect availability and partition tolerance characteristics of Swift, and this is the reason that Swift is categorized as an AP data store in term of the CAP theorem.

2.8 BFS Servers Interconnect

BFS servers use networking to access remote files on other BFS servers. BFS uses main memory to reduce I/O latency and to increase the read throughput. The networking between BFS servers should be fast and have low-latency. BFS is targeting environments where the network latency is small, and links are fast, such as data centers.

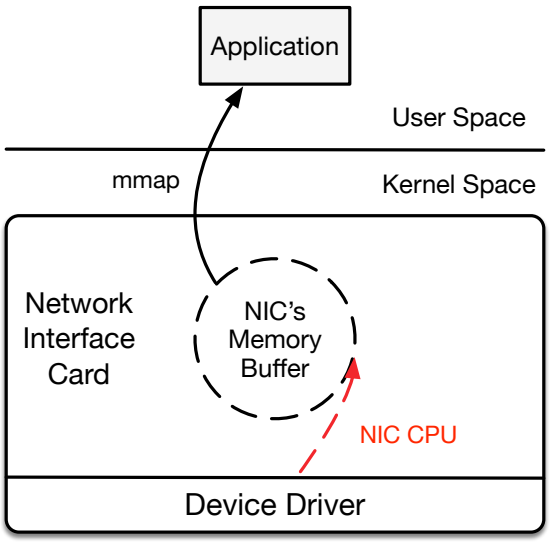


Figure 2.9: PF_RING Architecture

Using reliable sockets (TCP connections) provided by operating systems is the simplest and the standard way of transferring data between machines in a network. However, there is a notable overhead in processing each packet by the operating system network stack. Precisely, there are two main sources of delay in processing of each packet by the operating system kernel. First, the overhead of copying packets between the kernel and the user space. Second, the processing overhead of the network stack in the operating system. One solution to this problem is to use Remote Direct Memory Access (RDMA). RDMA allows accessing a remote machine's memory without intervention of its operating system by offloading the processing stack to the network card hardware. As a result, RDMA is able to provide high-throughput low-latency networking which is highly desirable in BFS. However, due to the lack of access to such equipment at the time of developing BFS, RDMA was not considered.

The third solution is to bypass the kernel network stack overhead. PF_RING [15] is a library capable of exposing the Network Interface Card's (NIC) buffer to the user space and manipulates this buffer directly without the involvement of the network stack. Using PF_RING BFS reduces the overhead of the network stack and avoids copying packets between the kernel and the user space. PF_RING is a packet capture library that provides an Application Programming Interface (API) to read/write directly from userspace to the NIC's buffer. Figure 2.9 depicts the architecture and the packet journey in PF_RING.

When a packet arrives, the special PF_RING device driver receives the packet in the DMA buffer. This buffer is shared by user space using memory mapped files. Therefore, applications can have access to the packets without any copying or overhead involved.

PF_RING reduces the overhead of the network stack and avoids copying between the kernel and user space, but it does not provide reliability guarantees offered by TCP. For instance, when the NIC's buffer is full, incoming packets will be dropped, and the application will not be able to process them. One of the main responsibilities of TCP is handling packet loss due to queue overflow at endpoints or routers. In contrast, BFS uses Ethernet Flow Control, specified in the IEEE 802.1Qbb standard [9], to create a reliable transfer protocol over raw ethernet. Ethernet Flow Control uses pause frames to stop the sender from sending more packets. Pause frames are sent before the NIC buffer becomes full. Pause frames request the sender to not send any packet for a specified period of time. Ethernet Flow Control is a hop-to-hop protocol which means a NIC can only send a pause frame to its immediate neighbours. Therefore, in order to create reliable communication along a path, all nodes should support ethernet flow. In addition, ethernet frames are not routable; consequently, BFS servers must be in the same broadcast domain to be able to communicate using PF_RING.

PF_RING needs customized drivers to be able to deliver packets to the user space without any copying. Therefore, PF_RING is not compatible with all commercial ethernet cards. In order to address this issue, BFS implements a TCP networking mode as well. In addition, the TCP mode can be used when BFS servers are not in the same broadcast domain. BFS networking mode can be configured using a configuration file. In the rest of this document when BFS uses PF_RING, it is referred to as the BFS_ZERO mode, and when TCP is used for network communication, it is referred to as the BFS_TCP mode.

2.9 Access Control in BFS

BFS uses libfuse to automatically handle file permission and ownership in the kernel on each BFS server. Therefore, BFS is not directly involved in handling file permissions and ownership information. However, ownership information on one BFS server may not be relevant on other BFS servers. Thus, remote files on each BFS server are created with the default user ID and group ID of BFS binary. Moreover, BFS does not implement Access Control Lists (ACLs). These limitation were introduced to make the implementation of BFS easier, and should be addressed in future releases of BFS.

Chapter 3

Related Work

As explored in previous chapters, there are different design perspectives that categorize different file systems. The access model divides file systems into two broad categories of local versus remote file systems. Local file systems serve data from a local device such as a disk, while remote file systems use networking to access data from a remote server. Remote file systems can be further categorized into shared file systems and distributed file systems. In a shared file system, all applications access and store data from/to a central file system server, whereas in a distributed file system data is stored and accessed from multiple servers.

In addition to the access model, the data model is another important design factor that categorizes file systems. Most traditional file systems, especially the local and shared file systems choose a flexible block-level data model, while the majority of recent distributed file systems use an object-level data model or even higher-level structured data models such as tables.

Finally, the name service design is also an important design decision that classifies file systems into different groups. Most local and shared file systems use a central architecture, while a distributed architecture is predominant in distributed file systems. In the rest of this section, various existing systems are discussed based on the aforementioned perspectives.

Local File Systems

Among different general purpose local file systems, tmpfs [39] is relevant because similar to BFS it uses main memory to store files. It provides a fully POSIX-compliant file interface over volatile main memory, and is useful for storing transient or easily recreated files such as

those that can be found in /tmp in Linux. Further, using non-volatile memory (NVRAM), persistency can be achieved in memory-based file systems. Many studies such as [30, 19, 28] try to design local file systems that are optimized for operating such devices. Although NVRAM devices are still very expensive to use in general workstations; they are becoming popular for server machines [23]. BFS can also utilize these devices. Using NVRAM allows BFS to operate with persistency without using any backend storage.

Most local file systems use a block-level data model; however, systems such as OBFS [45] use a object-based data model. OBFS is a local file system designed to store files as a set of objects on self-contained Object-Based Storage Devices (OSDs). OBFS stripes files across OSDs in form of objects, and tries to optimize disk layout to accommodate objects based on the workload. Evaluation results of OBFS demonstrate outperforming Linux Ext2 and Ext3 [44] by a factor of two or three, and it provides only slightly lower read performance and 10%-40% higher write performance than XFS [33].

Shared file Systems

In shared file systems, such as **NFS** [38] and AFS [26], all clients use remote servers to access files, and clients do not contribute to the storage. NFS utilizes Remote Procedure Call (RPC) for communication between clients and servers. The idea of using VFS was first initiated in NFS (cf. Section 2.4.3). NFS provides strong consistency and operates on byte-level semantics, it receives file system operations from VFS and forwards them with RPC to the NFS server side. The server side performs the operations and responses back to clients via RPC. NFS is a stateless protocol meaning that each RPC has all the necessary information to complete the call; therefore, crash recovery is very easy in NFS. For instance, if an RPC fails, a client sends another request until it receives a response from the server. Similar to NFS, BFS also uses a stateless protocol and failed requests are repeated until they receive a response from the server.

The main drawback of the NFS design is scalability. The central design of NFS limits its scalability with a large number of clients. AFS tries to alleviate some of these concerns by reducing server interactions (through caching mechanisms); however, the fundamental problem of having a central server design still exists. Some of AFS improvements are added to later versions of NFS, but due to the predominance of NFS in the marketplace, AFS has never gained the popularity of NFS.

Distributed file systems

There is a large body of work in distributed file systems. Most of these systems are developed to support reliability and scalability. One big group of these systems are object-based storage systems such Dynamo [22], Openstack Swift [14], and Cassandra [29]. All of these object storage systems relax consistency for availability and partition tolerance because reliability and availability are the most important requirements in environments they are used. For instance, in the case of Dynamo, any disruption will have a significant financial consequences [22]. In addition, they use object-based data models because most of the services that use these storage systems need a whole object access compared to a partial object access.

Dynamo uses consistent hashing for data partitioning, replication, and creating its flat namespace. Membership and failure detection are done using a gossip-based protocol. Dynamo only provides a simple object-based *get()/put()* interface. This is because services that Dynamo is built for do not require additional complex operations as the case in RDBMSs. Dynamo provides availability and scalability by using optimistic replication. In optimistic replication, changes in data items are propagated in an asynchronous manner in the background resulting in conflicting replicas. Due to the fact that Dynamo aims to be always writeable, it resolves conflicts during read operations, which guarantees writes are never refused. A unique design factor in Dynamo is the conflict resolution mechanism, which unlike many other data stores is delegated to clients instead of the data store.

Load balancing and the namespace in Dynamo are handled by partitioning data items over nodes using consistent hashing. Each data item is recognized by an id (using MD5 sum of its key) and is assigned to a node (coordinator) that is responsible for the corresponding portion of the hash space. Each data item is replicated at N (a configured parameter) nodes called preference list, including the coordinator for that data item, and N-1 successor nodes in the hash space. In contrast, it has been shown [36] that consistent hashing does not perform optimally in keeping the load variance low. This issue is further explained in Section 5.

Openstack Swift inherits many of Dynamo design decisions and differs only in small details such as membership/failure detection or the provided object interface. Membership and failure detection in Swift are done manually and there is no automatic mechanism to handle node churn. Further, Swift provides a more comprehensive object interface, whereas Dynamo only provides a simple *get()/put()* interface.

Cassandra also shares many similarities with Dynamo. Similar to Dynamo, Cassandra tries to provide a highly scalable and reliable storage facility. Conversely, it uses a somewhat

higher-level data model than Dynamo. Cassandra's data model is not neither object-level, nor a fully structured data model such as a table in relational databases, instead it supports dynamic control over the data layout and format. In fact, each table in Cassandra is a multidimensional map that is indexed by a key. Cassandra's API is simple and includes only three methods, *insert()*, *get()*, and *delete()*.

Similar to Dynamo, Cassandra uses consistent hashing for partitioning of data across clusters to serve availability. Cassandra uses different policies such as *Rack Aware* and *Datacenter Aware* to decide where to replicate data items at. Cassandra uses *Scuttlebutt*, a very efficient anti-entropy gossip-based mechanism to disseminate membership information and system related control states. Cassandra uses the so-called ϕ *Accrual Failure Detector* to detect node failures. These detectors are fast, accurate and adjust according to the server load conditions.

When it comes to latency and performance, Cassandra diverges from Dynamo and Swift. Cassandra is designed to handle millions of writes per day while Dynamo and Swift focus on availability even with high write latency. Therefore, Cassandra employs various techniques to improve the write latency and the throughput. A write in Cassandra includes a write into a commit log and an update into an in-memory data structure, which will be performed only after the successful write into the commit log file. Cassandra can be configured to perform either synchronous or asynchronous writes depending on applications' requirements. A read operation first queries the in-memory data structure before looking into files on disk.

Tachyon [31] and RamCloud [35] are examples of in-memory distributed file systems. Similar to BFS, these two systems use main memory for storing data. In the following, similarities and differences of these two systems with BFS are discussed.

Tachyon tackles the problem of slow I/O operations in big data computations. Similar to BFS, Tachyon uses a block-level data model. Unlike many systems, Tachyon does not use main memory for keeping hot data (cache behaviour) but instead replicates data asynchronously after it is written to main memory. Tachyon uses applications' hints and so-called linage concept [47] to recompute any loss in data.

Tachyon design is inspired by characteristics of big data workloads, such as deterministic computation (applications code is always deterministic), program size vs. data size (programs are small and replicable). These characteristics make Tachyon a viable system, but in the absence of any of these characteristics, Tachyon either fails, or provides no improvement. Tachyon stores a working set in the main memory and replicates new data asynchronously and recomputes them in case of loss using the lineage information. Implementation of Tachyon reported that it can attain write throughput of 300X higher

than Hadoop File System (HDFS).

Although Tachyon might be a valid and elegant solution for certain workloads, its limitations do not allow to use it for many applications such as HPC applications. In addition, Tachyon does not provide a general purpose file system interface and is highly customized for the Hadoop ecosystem.

Similar to BFS, **RAMCloud** utilizes main memory to store data. However, unlike BFS, RAMCloud's data is entirely kept in main memory. RAMCloud uses replication to achieve durability. However, this approach is costly because main memory is expensive, and this increases the amount of required memory depending on the number of replicas. On the other hand, using DRAM as backup is not very reliable, because a power outage might take down all machines including backup machines. To address this issue, RAMCloud utilizes buffered logging that uses both disk and memory for replication. In this approach, a single copy of an object is kept in the DRAM of a primary server and a copy is also kept in the DRAM of two (or more) servers temporarily until data is written on the disk. The important point is that the backup copies are not updated synchronously with the write operation. Instead, the primary server updates its DRAM and forwards log entries to the backup servers. This write operation returns as soon as the log entries are written to the DRAM of backup servers. Finally, backup servers write logs from their DRAM to disk. Although buffered logging allows read and write at DRAM speed, the overall system throughput is still limited by the disk bandwidth for writing the log entries. Thus, if a high write throughput is desirable, then the only way is to keep replicas in DRAM.

Another different design decision in RAMCloud is the data model. RAMCloud proposes an intermediate approach where the system does not impose structure on data but supports aggregation and indexing. In this approach, storage entities are objects that can be grouped and indexed.

Ousterhout et al. [35] claim that due to the low latency of write operations in RAMCloud, keeping strong consistency is possible. However, they do not describe the necessary details about how consistency is achieved and to what degree it is supported in RAMCloud. One can clearly see that strong consistency is not provided by RAMCloud. For example, in the buffered logging mechanism, if the primary node that holds a data item crashes before its backup server finishes writing to the disk (backup servers also fail to finish the write), the latest version of that data item is permanently lost while the write operation has already returned to the client. But it is assumed that data center wide applications usually do not require strong consistency, and because the latency of operations are extremely low in RAMCloud, lost data items can be quickly recomputed.

Ceph [46] and GlusterFS [6] are examples of POSIX-compliant distributed file sys-

tems. Ceph uses an object-based data model and utilizes OSDs. OSDs allows file systems to read/write arbitrary byte ranges from/to disks without handling the low-level block allocation details and granularities concerns. However, OSDs are not scalable due to little or no distribution of metadata. Ceph decouples data and metadata operations. It uses an object storage cluster for storing data and metadata information; further, it employs a separate cluster for handling metadata operations such as namespace operations (e.g. file creation, deletion, or rename). Ceph tries to achieve high scalability and performance by decoupling data and metadata. BFS is similar to Ceph in the sense that BFS also uses a separate cluster (ZooKeeper cluster) for handling namespace operations. However, unlike Ceph, BFS does not devote all metadata operations to a separate cluster. For example, opening and closing files is handled by BFS servers (storage servers) rather than ZooKeeper nodes.

Ceph provides a range of different interfaces. Ceph supports a POSIX-compliant interface as well as an object-based REST API. Ceph maps each file to an object and relies on the fact that OSDs allow byte-level operations without involving in the low-level block allocations. This is similar to what BFS does, but unlike OSDs, most object storage systems do not support byte-level operations. Ceph uses similar techniques to those of consistent hashing for handling the namespace. It hashes individual directories and dynamically re-partitions them when they become large. This scheme can be useful in the current BFS namespace design as well. For instance, it is possible to shrink those directories at ZooKeeper that pass a threshold. However, a more detailed plan about handling the namesapce scalability in BFS is provided in Section 5.

Ceph is a CP(Consistent and Partition Tolerant) storage system. This is because Ceph provides a fully POSIX-compliant interface; therefore, it needs to support strong consistency. On the contrary, Ceph has support for relaxing consistency promises to gain availability.

GlusterFS is a popular POSIX-compliant distributed file system. GlusterFS uses a block-level data model to store files. In fact, GlusterFS is not directly involved in storing files on disk. GlusterFS piggybacks on the existing local file systems such as ext3,ext4, or XFS, which is similar to the backend storage in BFS. In fact, GlusterFS is more of a shared file system than a distributed file system. It is similar to NFS in the sense that it aggregates the storage of commodity servers with a unified namespace and provides a remote access protocol to the files. However, GlusterFS shares many distributed file systems characteristics. For instance, similar to many distributed file systems, GlusterFS uses a distributed hashing mechanism know as *Elastic Hashing* to create its namespace. In fact, GlusterFS does not store any namespace metadata. It maps file names to storage servers using a hash function. Therefore, using this hash function each GlusterFS server

can find which server is storing which files. Further, to address the problem of adding and removing servers, GlusterFS uses a virtualization layer in which file names are mapped to virtual servers. Virtual servers are assigned to multiple physical servers using a seperate process. Therefore, in the event of adding/removing servers the hashing algorithm does not need to be changed, and virtual volumes are reassigned or migrated using a seperate process.

The main goal of GlusterFS is to integrate easily in different environments. For instance, GlusterFS provides several interfaces including, a POSIX-compliant file interface through the FUSE library and the NFS protocol. Further, it supports several language bindings including a C API that BFS uses to interact with as the backend storage.

As stated in Section 2.7.2, GlusterFS sacrifices availability to provide strong consistency, but similar to Ceph, GlusterFS can be configured to relax its consistency model to gain availability in return.

Table 3.1 summarizes storage systems that are discussed in this chapter.

System	Interface	Data Model	Namespace	CAP	Membership Detection	Failure Detection	Replication
BFS	POSIX-Similar	Block level	ZooKeeper	Depends on the backend	Zookeeper watches	ZooKeeper Ephemeral nodes	Using backend
Dynamo	API-based	Object-based	Consistent Hashing	AP	Gossip-based	Timeout based	Replicated at N nodes
Cassandra	API-base	Structured object-based	Consistent Hashing	AP	Gossip-based	ϕ Failure detector	Replicated at N nodes
RAMCloud	Table-based	Object-based	Central index-based	AP	Not specified	Not specified	1 Replica in DRAM and 2 on disk
Openstack Swift	API-based	Object-based	Consistent Hashing	AP	Manually	Timeout based	Replicated at N nodes
Tachyon	API-based	Block-level	Not specified	Not specified	Not specified	Not specified	Asynchronously
GlusterFS	POSIX-compliant	Block-level	Elastic Hashing	CP	Manually	Automatic	Replicated at N nodes
Ceph	POSIX-compliant with Object API	Object-level	Consistent Hashing	CP	Automatic	Automatic	Replicated at N nodes

Table 3.1: Summary of different storage systems

Chapter 4

Evaluation

As explained in Section 1, BFS is a combination of in-memory and remote file systems. BFS is expected to deliver similar performance to an in-memory file system when data and applications are co-located, and performs comparable to remote file systems otherwise. The main purpose of evaluations is to understand whether a simple design such as BFS is successful in achieving its goals while performing comparable and/or superior to other more complex solutions. Four groups of experiments are conceived to evaluate BFS. The first experiment is dedicated to measuring the performance of BFS and understanding if it delivers the expected performance under different scenarios and if comparable to other solutions. In the second group of experiments, the reliability and failure tolerance of BFS is evaluated to find out whether BFS can efficiently recover from failures or not. Further, the scalability of BFS is measured with various synthetic workloads to determine if BFS can scale with an increasing number of clients. Finally, individual microbenchmarks are used to evaluate BFS_ZERO mode and measure if it is successful in decreasing the network latency.

There are many benchmarks that have been developed, either by industry or academia, to measure different aspects of a file system. However, most of them focus on a single dimension of a file system and are developed for evaluating a specific file system. In most cases, researchers choose to use their own ad-hoc benchmark [42]. Table 4.1 shows the results of a study by Tarasov et al. [42] between 1999-2010 on 168 file system centric research papers from major computer science conferences reporting how many times each benchmark had been used.

Unfortunately, it can be seen from this study that there is not a widely accepted consensus among researchers on file system benchmarks. As a result, it is necessary to

Benchmark	Benchmark Type					Used in papers	
	I/O	On-disk	Caching	Meta-data	Scaling	1999-2007	2009-2010
IOmeter	•					2	3
Filebench	•	○	○	○	•	3	5
IOzone		○	○		•	0	4
Bonnie/Bonnie64/Bonnie++		○	○			2	0
Postmark		○	○	○		30	17
Linux compile		○	○	○		6	3
Compile (Apache, openssh, etc.)		○	○	○		38	14
DBench		○	○	○		1	1
SPECsfs		○	○	○	•	7	1
Sort		○	○		•	0	5
IOR: I/O Performance Benchmark		○	○		•	0	1
Production workloads	⋆	⋆	⋆	⋆		2	2
Ad-hoc	⋆	⋆	⋆	⋆	⋆	237	67
Trace-based custom	⋆	⋆	⋆	⋆		7	18
Trace-based standard	⋆	⋆	⋆	⋆		14	17
BLAST		○	○			0	2
Flexible FS Benchmark (FFSB)		○	○	○	•	0	1
Flexible I/O tester (fio)	○	○	○		•	0	1
Andrew		○	○	○		15	1

Table 4.1: Benchmarks Summary. • indicates the benchmark can be used for evaluating the corresponding file system dimension; ○ is the same but the benchmark does not isolate a corresponding dimension; ⋆ is used for traces and production workloads. Taken directly from [42]

evaluate different aspects of a file system using different techniques, instead of using a single tool or benchmark.

BFS is a different storage service than regular local or distributed file systems. BFS is a combination of local in-memory file systems and remote file systems. Therefore, as stated before, the goal of this chapter is to understand the performance of BFS comprehensively rather than competing with other file systems. However, in order to put the results in perspective, it is necessary to report the results of the same experiments with an existing system as well. As stated in Chapter 3, there are distributed in-memory storage systems such as Tachyon [31] or RAMCloud [35]; however, Tachyon is specifically designed for the Hadoop environment and provides a Java API, rather than a general purpose file interface. Similarly, RAMCloud does not include a POSIX-like file interface and provides a high-level structured data model. On the other hand, there are many other disk-based distributed storage systems, such as Dynamo [22], Openstack Swift [14], or Cassandra [29], which provide an object-based data model rather than files. Finally, POSIX-like distributed file systems, such as GlusterFS [6], Ceph [46], NFS [38], or Lustre [13] are suitable candidates

for comparison. GlusterFS is chosen for two reasons. First, it is a widely used system in industry and is easy to use. Second, BFS includes a GlusterFS backend plugin; therefore, it is informative to compare BFS with GlusterFS because both systems store their files permanently in the same backend setup.

4.1 Testbed Environment

All the evaluations of BFS are run in a cluster of 16 machines at the University of Waterloo. The cluster contains one head node and 15 compute nodes. The configuration of head node and compute nodes are summarized in Appendix B. One head node and three compute nodes are used to build the backend storage.

GlusterFS is deployed as the backend storage solution for BFS. GlusterFS provides three different access methods, FUSE-based file interface, NFS, and a C file API (libgfapi). BFS uses libgfpai to synchronize data with the backend, and the two other clients (NFS and FUSE) are used for GlusterFS evaluations. However, in all experiments the result of the GlusterFS client which performed better is considered as the GlusterFS result. A GlusterFS volume is initiated on these four nodes using their SATA3 Solid State Disk (SSD) drives with a total capacity of 1 TB and replication of one. All SSD drives are formatted using the XFS file system. It is worthwhile to mention that the same GlusterFS deployment is used for GlusterFS measurements.

In addition to the backend nodes, three other nodes of the cluster are used to create a ZooKeeper deployment. ZooKeeper is a quorum-based consensus protocol which means that to tolerate F node failure, it needs $2 \times F + 1$ nodes. Therefore, this setup is capable of tolerating one node failure. A failure is considered as a node crash, or any error in the network that partitions a node from the majority of nodes. The latest version of ZooKeeper, 3.5.0-alpha is used, and ZooKeeper instances are configured to use SSD drives.

Using four nodes for the backend storage and three nodes for the ZooKeeper deployment leaves nine nodes for BFS. Each of these nodes has 64 GB of RAM; therefore, theoretically BFS can have an aggregated capacity of 576 GB. However, BFS is configured to use 90% of each node memory; leaving the rest to the operating system and/or other running programs. This gives BFS an aggregated capacity of approximately 520 GB with an approximate maximum file size of 57 GB.

In all experiments, BFS is used in TCP mode (unless explicitly stated) due to the lack of support for Mellanox 10 Gb/s ethernet adaptors in the PFRING library. However thanks to the PFRING community, a research license was acquired for Intel i350 Gb/s

ethernet adaptors, and the results of comparing BFS in TCP and ZERO mode using Intel cards are presented in Section 4.5.

4.2 Throughput

A crucial evaluation for a file system is to report the maximum throughput it can deliver. The main goal of the measurements in this section is to provide an overview of the maximum read/write throughput that BFS can achieve and compare it against GlusterFS in different scenarios. Normally, it is expected when application and BFS servers are co-located that reads are performed at the memory speed similar to an in-memory file system. On the other hand, when reads are served through remote BFS servers (duel to the lack of space on a BFS server) they are limited by the network bandwidth and remote servers' memory. Further, similar to remote file systems, writes are limited by the network bandwidth and the backend storage. As a result, it is anticipated to see a range of different throughput results depending on the usage scenario.

The *Iozone* [11] benchmark is used to measure the read/write throughput. Iozone is a popular file system benchmark used to measure a variety of file operations including meta-data operations and aggregated read/write throughput. All nine nodes in the cluster are instructed to run BFS with 90% of their available physical memory, and Iozone is run on one of them.

The workload size is crucial in this experiment. For instance, if the workload size is less than each server's capacity, then in the case of BFS, neither writes nor reads are forwarded to other servers, and reads are all served from local memory. Thus, in order to cover various scenarios, four different workload sizes are specified, 32 GB (half of a single server memory), 60 GB (BFS maximum capacity per server), 128 GB (two times of the BFS maximum available space per server), and finally 256 GB (four times the size of the BFS maximum available space per server). The following Iozone command is used on one of the servers and is repeated 10 times for BFS, GlusterFS FUSE, and GlusterFS NFS as well:

```
./iozone -c -e -i 0 -i 1 -+n -r 16M -s [SIZE] -t [THREAD]
```

In this command, SIZE represents the file size, and THREAD determines the number of concurrent files to be written. For example, for 32 GB, the size is set to 1 GB and the number of threads to 32 and similarly for other workset sizes. In addition, the *-c* and *-e*

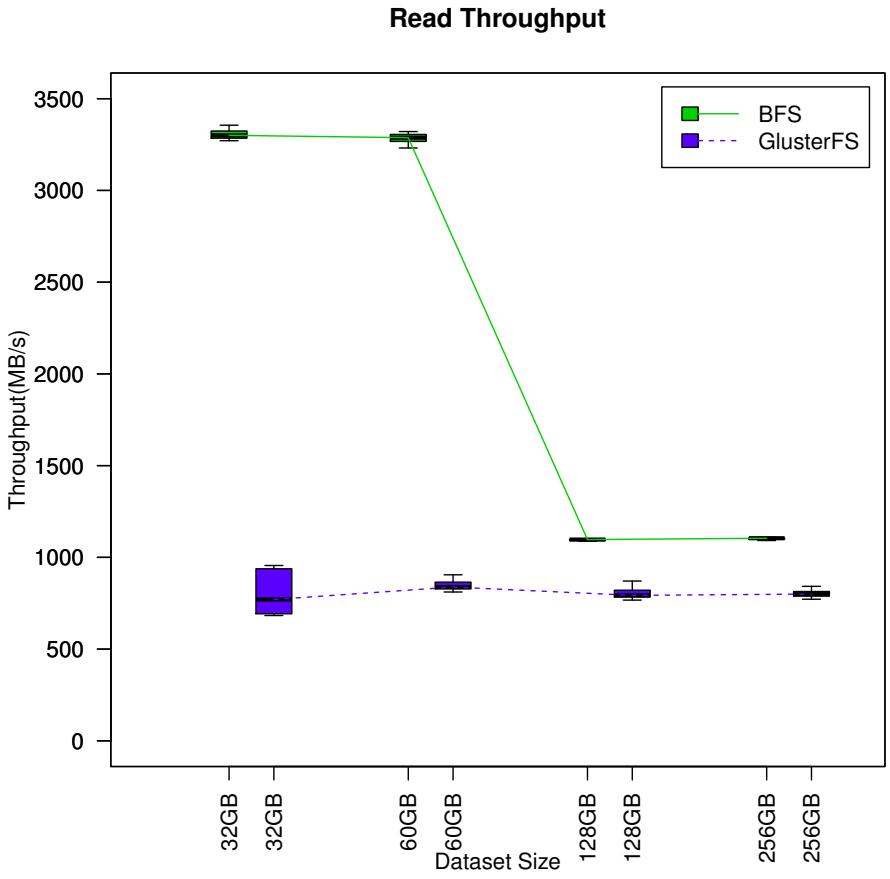


Figure 4.1: Read throughput evaluation

options instruct Iozone to include the time of *fsync* and *close* system calls in the measured time. Figure 4.1 shows the boxplot of read bandwidth for BFS and GlusterFS.

As expected, when the workload fits in the memory of a single server, reads are performed at the rate of memory for BFS; therefore, in this case BFS serves reads as an in-memory file system. On the other hand, when the total workload size passes the memory limit of a single server (60 GB), BFS is still faster than GlusterFS because, although BFS read requests are served through the network (similar to a remote file system), they are being read from remote servers' memory, compared to GlusterFS that serves reads from the backend servers' disk. In fact, in the case of BFS, remote reads are limited by the network bandwidth, whereas in GlusterFS they are limited by the network and the

backend storage.

One notable point about the workload sizes 128 GB and 256 GB is that some of the requested files should be served from the memory of the server on which the requested files are first written; therefore, the read rate should be a mix of memory and network rate. However, in practice all files are served remotely because of fast concurrent writes by Iozone. Iozone concurrently writes all the requested files until the capacity of a server is reached. At this point, BFS starts to move one of the files to another node. Meanwhile, other concurrent writes trigger moving other files because moving a file is a lengthy process, and before a file is moved to create free space for other writes, other writes trigger move for other files. As a result, after reaching the memory limit in this case, almost all read requests are served remotely due to the current moving policy in BFS, Section 2.4.5.

One question is why at 32 GB, GlusterFS does not use the Linux page-cache at all while the workload fits in the memory of a single server. GlusterFS avoids using the Linux page-cache to provide strong consistency. For instance, consider that server A in GlusterFS opens file F and writes X into it. In addition, assume X has not yet been committed to the backend storage of GlusterFS. Server A performs a read on F and sees X; meanwhile, server B also opens file F and performs a read on it and sees an older version of F (not X). Therefore, strong consistency is violated in this example. In order to avoid such consistency issues, GlusterFS mounts its clients with the **direct_io** flag, which instructs GlusterFS to bypass the kernel page-cache altogether. Hence, avoiding page-cache in GlusterFS is why the read throughput is not at the memory speed even when the total file size is less than the memory capacity of a server.

Figure 4.2 presents the write throughput for BFS and GlusterFS. It is anticipated that GlusterFS and BFS have a similar write throughput since both are writing to the same backend storage (GlusterFS server). It can be seen that when the workload fits in the memory of the server that Iozone is running on, the write throughput of BFS and GlusterFS are very close. However after 60 GB, the BFS write throughput slightly declines and then becomes steady. There are two main reasons for this behaviour. First, after passing the memory limit of a server, writes are routed to remote servers in BFS, and this results in extra latency. Second, as explained for read throughput, Iozone writes files concurrently, meaning that once the memory of a server is full, the partially written files start to be moved to other remote servers, and this has a significant overhead.

In conclusion, although this benchmark is not fully representative of all real scenarios, it shows how BFS throughput is alternating between an in-memory file system and a remote file system throughput depending on the scenario. In fact, this experiment confirms the design and hypothesis of BFS as a combination of in-memory and remote file system.

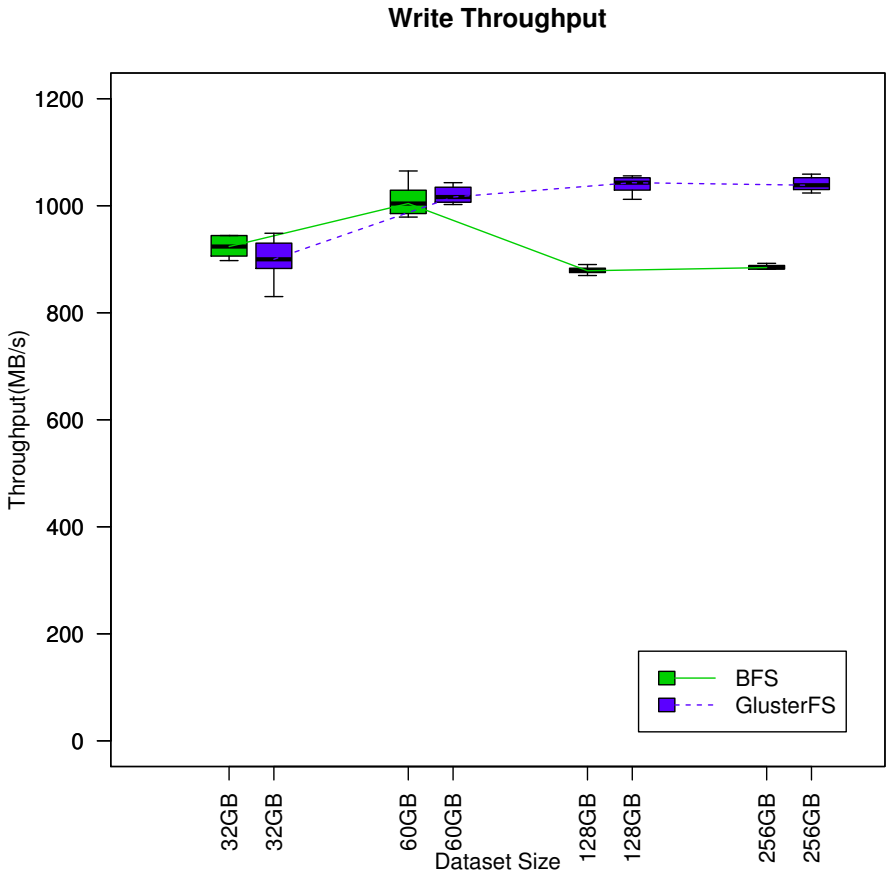


Figure 4.2: Write throughput evaluation

4.3 Reliability

BFS is different than in-memory file systems in the sense that it is able to survive failures; therefore, it is essential to evaluate the reliability of BFS and how efficiently it recovers from failures. In this section two sets of experiments are described to measure reliability and recovery overhead of BFS. First the efficiency of *fsync* system call is measured as the mechanism to achieve persistent and durable writes. Seconds, the recovery latency of BFS is measured after a crash has happened.

4.3.1 *fsync* Latency Test

In BFS, *fsync* or synchronized writes flush the in-memory data to a persistent backend storage and this is what differentiates BFS from an in-memory file system. In fact, the fsync latency represents the vulnerability window of system since the write are not durable before they are flushed to the backend storage. It is expected that BFS and GlusterFS clients have roughly the same fsync latency because they all write to the same backend. The *fsync-tester* [4] benchmark is used to measure the latency of fsync system calls in BFS and GlusterFS clients. The fsync-tester benchmark is a simple tool that writes 1 MB to a pre-allocated file (the same file, the same offset and the same size) and invokes fsync on that file and measures the time for fsync (write time is calculated separately). It repeats the same scenario every one second. 1000 data points for BFS, GlusterFS FUSE and GlusterFS NFS are collected and presented in Figure 4.3. An important observation from this figure is the high latency of GlusterFS clients compared to BFS. Normally, it is expected for BFS and GlusterFS clients to have the same latency because they all write to the same GlusterFS backend server; especially BFS and GlusterFS FUSE clients both use the FUSE library. In contrast, the BFS latency is approximately three times less than that of GlusterFS clients. An investigation revealed that the GlusterFS FUSE clients use a fixed 4 KB block size to flush the write-behind buffer (1 MB by default in GlusterFS) that results in a significantly high latency. In contrast, BFS uses a big block size of 16 MB.

Similar to the GlusterFS FUSE client, the GlusterFS NFS client also uses a small block size (a varying block size between 4 KB and 32 KB), but unlike the FUSE client, the NFS client does not use a write-behind buffer, and it forwards all writes to the backend storage. As a result, the fsync system call should have a very small latency in the GlusterFS NFS client because all writes are already transferred to the backend, and there is no buffer to be flushed. On the other hand, GlusterFS NFS's daemon in the backend uses a write-behind cache, and an fsync system call on the client triggers the write-behind buffer on the backend server to be written to disk, which in turn causes the observed high latency for the GlusterFS NFS client. Another observation in this graph is the high jitter in the GlusterFS FUSE client that is unclear in its origin.

4.3.2 Crash Recovery Evaluation

In addition to measuring the efficiency of *fsync* as the mechanism to achieve reliability and persistency, it is important to evaluate how efficiently BFS can handle failures. In essence, there are three elements that might fail in a BFS deployment. First, a BFS server, second, a ZooKeeper node, and finally, a backend node. ZooKeeper is a highly available and fault

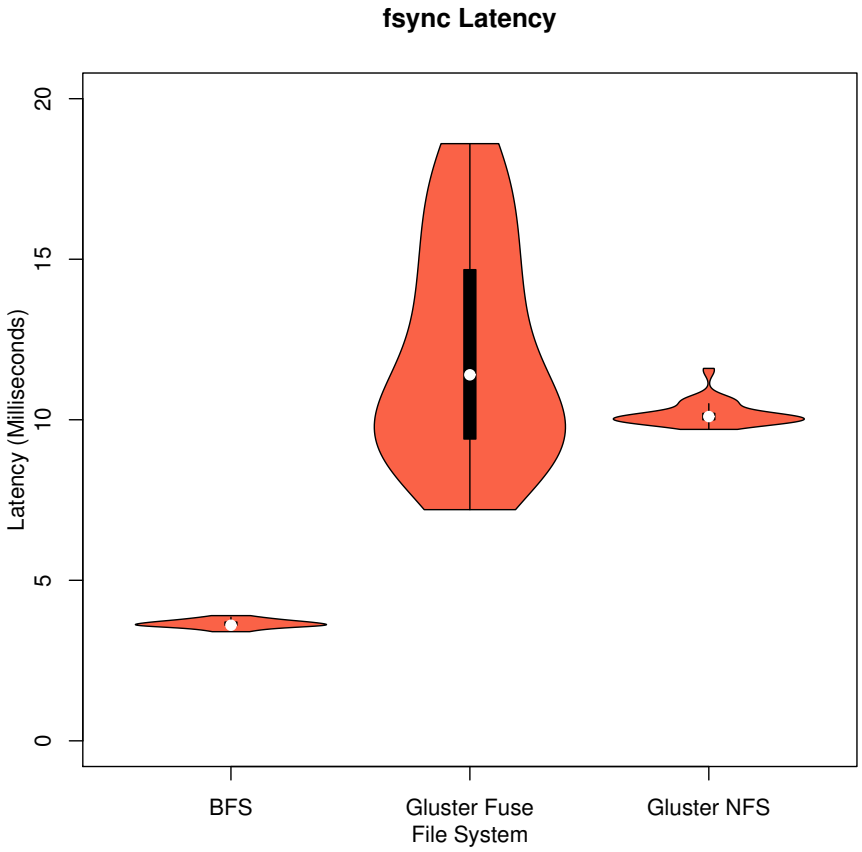


Figure 4.3: *fsync* Latency Test

tolerant system; it uses replication and various mechanisms to survive failures. Evaluation of failure recovery in ZooKeeper is out of the scope of this thesis, but is discussed in the original ZooKeeper paper [25]. Similarly, the evaluation of backend node failures is omitted in this thesis because it is a characteristic of the backend storage.

BFS server failure can be a temporary network partitioning, power failure, or any other reason that isolates a server from the rest of servers. As explained in Section 2.4.7, membership and failure detection in BFS are done by ZooKeeper watches; therefore, evaluation of these mechanisms is skipped. The only question left to explore is that how efficiently BFS recovers after detecting a BFS server failure. An experiment is designed to measure how fast BFS recovers files that are hosted at the crashed servers. In this experiment, first,

all nine BFS servers are filled to half of their capacity with random files, and then one, two and four of these servers are randomly crashed. The elapsed time since the crash time up to when all missing files are recovered in other servers is measured using crash logs. This experiment is repeated 10 times, and the violin plot of the recovery time is depicted in Figure 4.4.

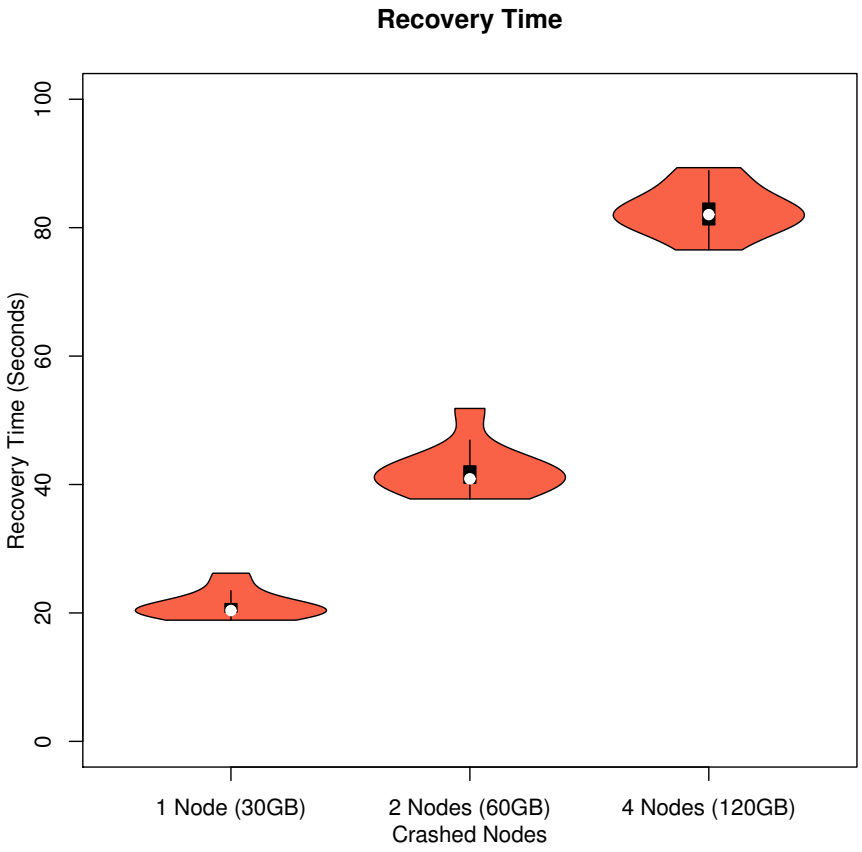


Figure 4.4: Recovery Time in BFS with a varying number of crashed servers

It can be observed from Figure 4.4 that the recovery time is increasing linearly with an increasing number of crashed servers. This means that the recovery rate is limited by the backend storage and the network bandwidth. Therefore, increasing these two limits can improve the recovery rate in BFS. By dividing recovery time over the size of crashed files in each case, an average rate of 1500 MB/s is achieved, which is close to the theoretical

maximum throughput of the backend storage in the experiment setup. Moreover, in all cases, there is a very small jitter in the recovery time that indicates the only factors which affect recovery time in BFS are the network and the backend storage bandwidth.

4.4 Scalability

An important aspect of a file system is how it behaves under different workloads with an increasing number of clients. The SPEC SFS 2014® benchmark [18] is used to study the average latency of BFS file operations under different workloads with a varying number of clients. The SPEC SFS 2014 benchmark is the latest version of the Standard Performance Evaluation Corporation benchmark suite measuring file system throughput and response times. SPEC SFS facilitates a standard way of comparing performance across different platforms. SPEC SFS includes four different workloads, which cover a mixture of file metadata and data oriented operations. The list of file operations covered by SPEC SFS is shown in Appendix C. Moreover, SPEC SFS supports multiple clients and is a distributed application that coordinates and conducts testing across all nodes that are used to test a storage service [18].

SPEC SFS uses *Business Metric* as the unit of workload. Business metric is an independent unit of workload that represents how many instances of the specified workload are running concurrently. For example, for the DATABASE workload of SPEC SFS, which represents the typical behaviour of a database, a business metric of three means that three independent databases are using the underlying file system concurrently [18]. Load scaling is also achieved by increasing the business metric in SPEC SFS. In fact, the amount of jobs each load or business metric is performing is fix; however, increasing the business metric will add more work. For example, if the DATABASE workload performs 16 ops/sec, increasing the business metric to 2 will impose 32 ops/sec on the underlying file system. In addition, SPEC SFS uses *Success Criteria* to decide if a workload was successfully performed or not [18]. Success criteria is a latency threshold that defines what percentage of the committed operations of a workload should have been succeeded for a workload to be considered successful or failed.

SPEC SFS has four execution phases. The first phase is *Validation* that tests if all required I/O operations are supported in the underlying file system. The second phase is *Initialization* that writes workload files to the underlying file system. The third phase is *Warm-Up* and finally *Run* phase that performs measurements and reports results back to the user. BFS uses **direct_io** flag of FUSE to omit Linux kernel page-cache altogether; therefore, no caching effect is affecting BFS. However, for GlusterFS, all the default mount

options are used, and no mechanism is employed to stop GlusterFS from using Linux page-cache.

In all SPEC SFS workloads, the default runtime and warm-up times of 300 seconds are used. Appendix D contains the detailed configuration file used for running SPEC SFS 2014. Further, the business metric is increased to fill the maximum storage capacity of the system (500 GB).

4.4.1 Usage Scenarios

Before presenting results of the SPEC benchmark, it is important to discuss different scenarios in which BFS might be deployed. There are two main scenarios in which an application might use BFS, balanced versus unbalanced. In order to understand these scenarios, consider a group of BFS servers and an application running on these servers. In the first scenario, the application is balanced across these servers (balanced scenario) and most requests are served from local BFS servers (served from memory). In the second scenario, unbalanced scenario, one of the instances of applications is doing very data intensive operations while others are not, and this leads to a more remote (served from other BFS servers through networking) access pattern.

The balanced scenario is the preferred usage scenario for BFS since it includes less remote accesses; however, the choice of BFS usage scenario is inherently tied to the environment where BFS is used. For example, in the big data computation or HPC environments, it is very likely that computation nodes are used as storage nodes (BFS servers) as well. Therefore, these environments naturally fit with the balanced scenario. As a result, for the SPEC SFS evaluations both scenarios are included.

The simplest way to model this scenarios in the evaluations is to place the workload generator on a single BFS server for the unbalanced scenario, and use the SPEC SFS distributed workload generator on multiple BFS servers for the balanced scenario. Similar to BFS, GlusterFS can also be used in a balanced or unbalanced scenario. However, because GlusterFS clients do not store anything at individual servers, and all requests are served directly from the backend storage, GlusterFS in both scenarios performs similarly.

4.4.2 SPEC Workloads

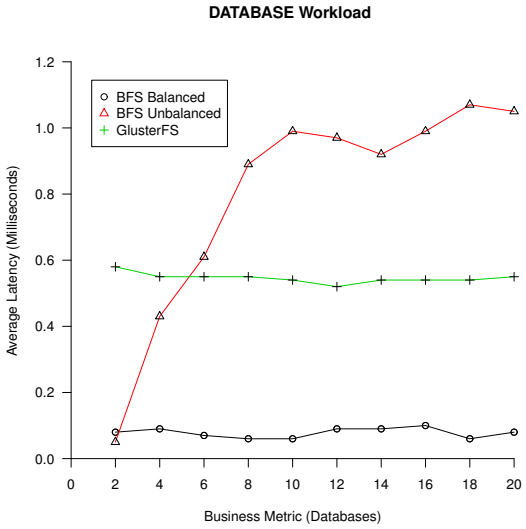
Figure 4.5 presents the results of four different SPEC workloads, DATABASE workload, which represents the behaviour of a typical database system; Video Data Acquisition

(VDA) workload, which simulates storing data from a temporary source such as video surveillance cameras; Virtual Desktop Infrastructure (VDI) workload, which emulates interactions of a hypervisor and a storage system when the virtual machines are running on ESXi, Hyper-V, KVM and Xen environments; and finally, Software Builds Systems (SWBUILD) workload, which tries to mimic the behaviour of large software build systems [18].

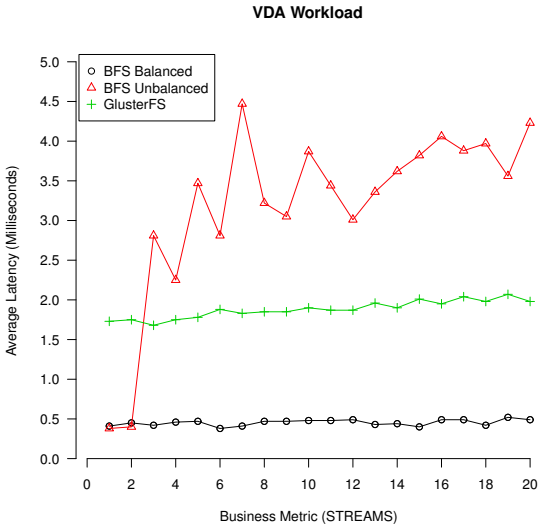
In DATABASE, VDA, and VDI workloads, as soon as the workload passes a single server's memory capacity, and remote operations are involved, latency significantly increases for BFS. As mentioned in Section 4.4.1, this is on the grounds that all operations are routed through a single BFS server (unbalanced scenario). For example, for a write task to finish, it should first be forwarded to the server that is hosting that file (the remote server), and then once it is written in the memory of the remote server, it will be flushed to the backend storage. However, even considering this extra one hop routing overhead, BFS latency in the unbalanced scenario is still comparable to GlusterFS. On the other hand, in the balanced scenario, increasing the business metric does not affect BFS latency, and BFS provides a significantly low and steady latency because most requests are served from the local servers memory.

SWBUILD workload is different form other workloads in the sense that it creates and modifies a large number of files. This workload is derived from traces of software builds on hundreds of thousands of files. Usually in build systems, file attributes are first checked and then if necessary the file is read, compiled and written back as an object file. This workload has only one build component and launches five processes per business metric. Each business metric in this workload creates about 600,000 files with a Gaussian distribution file sizes centered at 16 KB [18].

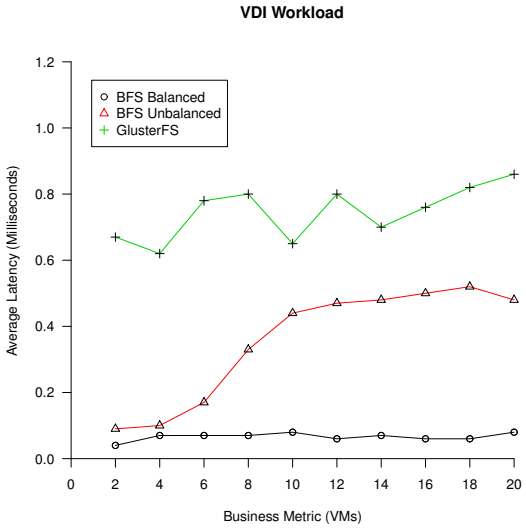
SWBUILD workload is important for BFS because as mentioned in Section 2.3, using ZooKeeper as the metadata server can become a bottleneck in case of workloads with many files. It can be seen from Figure 4.5d that BFS in unbalanced mode can only succeed the first business metric, and the balanced scenario completes only two business metrics with a significantly large latency. After two business metrics, the latency passes the success ratio of the workload, and the benchmark fails. The main reason is that the enormous number of files significantly increases the size of each BFS server's metadata directory at Zookeeper and crashes the ZooKeeper service. Therefore, this is one of the areas in which BFS needs to improve upon, and detailed plans and discussions are presented in Section 5. GlusterFS, as well as many other distributed file systems (Openstack Swift, Cassandra, or Dynamo), use consistent hashing [27] to keep track of which files (objects in case of object storage) are kept at which servers. As Figure 4.5d shows, GlusterFS scales and performs well with a large number of files.



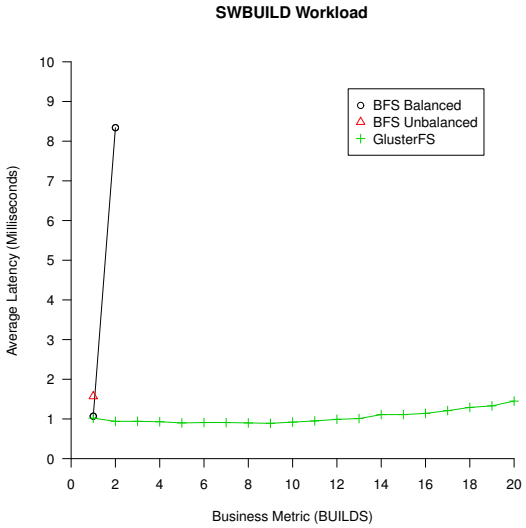
(a) Database workload



(b) Video Data Acquisition workload



(c) Virtual Desktop Infrastructure workload



(d) Software Build Systems workload

Figure 4.5: SPEC SFS workloads

SPEC workloads simulate a wide range of real workloads with a varying range of clients. It can be seen from results of these workloads that BFS latency in unbalanced scenario is comparable and similar to remote file systems such GlusterFS. This is expected since in the unbalanced mode, remote access pattern is the dominant access pattern. On the other hand, the balanced scenario indicates the superior performance of BFS due to having a dominant local memory access pattern. Finally, the SWBUILD workload strongly shows the limitations of ZooKeeper in handling large directory nodes which is one of the main areas for BFS to improve upon as future work.

4.5 BFS_ZERO

In this section two different modes of data transfer among nodes, BFS_ZERO and BFS_TCP are evaluated. Two sets of experiments are designed to compare the latency and the throughput of these two transfer modes. It is expected that BFS_ZERO provides a lower latency because it bypasses the regular network stack. In addition, BFS_ZERO should have a higher throughput because of the reduced packet overhead and the lower latency.

4.5.1 Latency comparison

In order to compare the latency of BFS_ZERO and BFS_TCP, the *ioping* [10] tool is used. *ioping* measures the I/O latency in a similar manner that *ping* measures the network latency. It sends read/write requests of a specified size (4 KB by default) at regular intervals (one second by default) and reports back the latency of the request. In order to avoid any effect of backend storage, BFS is configured to not use any backend storage. Therefore, writes and reads are performed entirely in main memory of a server. Moreover, it is necessary for operations to go through the network and be performed on a remote server. Therefore, two BFS servers are instructed to run BFS, one with capacity of zero (server A) and the other one with 90% of the available physical memory (server B). As a result, any I/O operation performed on server A is forwarded to server B. ioping is run on server A, with the following command:

ioping -c 1000 -W -s 4k

This command commits 1000 4 KB write requests at server A that will be routed to server B. The same command is used two times, once using BFS_ZERO transfer mode and once using BFS_TCP mode. In addition, an equivalent experiment is performed with the

network *ping* command to measure the network latency between server A and B. *ping* is performed from server A to B with a 4 KB packet size. It is expected that BFS_ZERO and *ping* have a similar latency because, although *ping* requests are served through the regular operating system network stack, it does not involve copying data between the kernel and the user space. Consequently, *ping* has a minimum processing overhead in the network stack as is the case for BFS_ZERO. Figure 4.6 shows the violin plot of requests' latency for BFS_ZERO, BFS_TCP and network ping with the packet size of 4 KB. A violin plot is similar to a boxplot with an additional overlaid curve that represents the probability density of data, which is useful to understand how scattered the data is. As expected, BFS_ZERO has a significantly lower latency than BFS_TCP and is very similar to *ping*. Moreover, it is interesting to see how scattered the latency is in case of BFS_TCP, while BFS_ZERO and *ping* have a latency that is highly centered around the mean of data, meaning a negligible jitter.

4.5.2 Throughput Comparison

In this experiment, the maximum throughput of BFS_ZERO is compared to BFS_TCP using a similar setup to the previous section. Again, the backend storage is disabled to prevent any external effect. Similar to Section 4.2, *Iozone* tool is used to measure the aggregated read/write throughput.

As expected, both BFS_ZERO and BFS_TCP utilized the link capacity (1 Gbit/s or 1000 Mb/s). However, BFS_ZERO has a slightly higher throughput than BFS_TCP due to the reduced packet header. BFS_ZERO only adds an eight bytes packet header (plus the ethernet header that is shared with BFS_TCP mode) yielding the following goodput (application level throughput):

$$\frac{1500(MTU) - 8}{1526(EthernetFramesize)} \times 1000Mb/Sec = 978Mb/s \text{ or } 122MB/s$$

A similar calculation for the theoretical goodput of BFS_TCP results in 120 MB/s. Therefore, it can be seen that both transfer modes provide a roughly similar goodput.

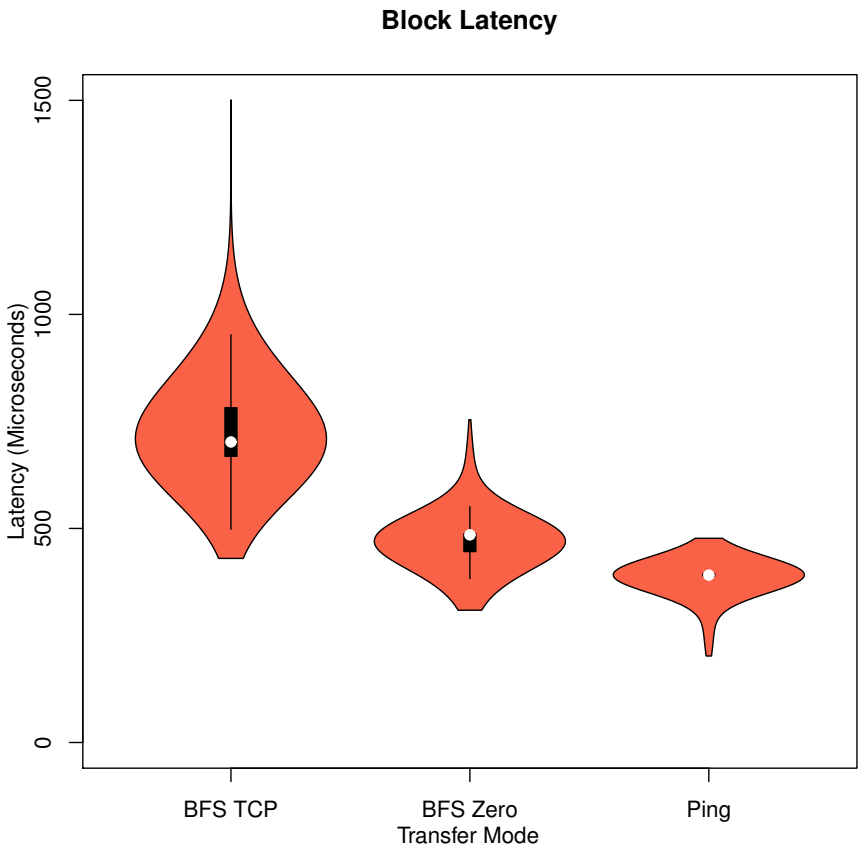


Figure 4.6: Latency for 4 KB blocks in microseconds

Chapter 5

Future Work

One of the main limitations of BFS is handling a large number of files. This stems from the ZooKeeper limitation in updating a large file. Many distributed storage systems such as Dynamo [22], Cassandra [29], Openstack Swift [14], or GlusterFS [6] use consistent hashing for organizing a shared namespace. Consistent hashing is a recent type of distributed hashing technique, which is popular among many P2P and cluster systems. It randomly divides the hash space to multiple parts that are assigned to different nodes in the system. Essentially this means that the hash space is divided to a lot more ranges than number of nodes, which reduces the variance.

The results of the SWBUILD workload in Section 4.4 indicate that the consistent hashing technique is effective and scalable to millions of files. However, a recent study [36] shows that although consistent hashing technique works well with a massive number of files, it is not very successful in keeping a low load variance among nodes. At runtime, consistent hashing should split the hash space more often to keep a fixed namespace load variance, but splits are expensive and complicated. The study presents GIGA+ [36] that solves this issue by using an optimized binary splitting mechanism. Evaluations of this mechanism indicate far fewer split than consistent hashing, yet lower load variance. As a future work direction, BFS is going to utilize similar techniques to those of consistent hashing and GIGA+ to approach scalability issues with a large number of files.

As explained in Section 2.4.5, BFS needs a hybrid and comprehensive mechanism to actively and proactively cope with the storage load variance problem. Each of the explained policies in Section 2.4.5 are effective under certain circumstances at the overloaded server, and none of them can always perform the best. Therefore, to choose the best policy, a comprehensive moving policy should evaluate the current status of servers as well as the

load in the system. In addition, the throughput evaluations in Section 4.2 also indicate the necessity of such a mechanism in BFS. Thus, a proper load balancing mechanism is one of the important topics that should be included in the future road map of BFS.

The choice of granularity in BFS affects many design decisions. As explained in Section 2.4.2, BFS uses a block-level granularity with a file as the storage entity because of simplicity; however, it might not be the best option. For instance, shrinking a file to multiple objects allows more efficient handling of overflows. Therefore, one area that needs to be studied further is the proper data model and granularity in BFS.

Another area that BFS can improve upon is how to react when all BFS servers reach their maximum capacity. The current implementation just treats this situation as a lack of space, but there are better ways to handle this situation. For instance, BFS can behave similar to a distributed cache, and when there is no free space at any server, it can swap out some existing files (based on a specified policy) in favour of new files. This feature can significantly improve the usability of BFS when the size of backend storage is massive, and BFS can not be a true mirror of the backend storage. In addition, although the cost per gigabyte of main memory has decreased recently, it is still significantly more costly than magnetic storage. For instance, BFS is a very costly choice for storing large data items such as images and videos. However, using the mentioned feature, BFS can be used as a distributed cache layer over a magnetic storage layer and speed up file accesses. Hence, considering BFS as a distributed file cache is a very promising future goal.

Finally, it is explained in Section 2.8 that RDMA is not utilized in BFS due to the lack of access to RDMA equipment. However, BFS can highly benefit from RDMA considering the low-latency and high-throughput networking it provides. In addition, RDMA is a hardware assisted solution and does not does not have limitations of BFS_ZERO mode. Therefore, it is necessary to add support for RDMA in future releases of BFS.

Chapter 6

Conclusion

BFS is a simple design which combines the best of in-memory and remote file systems. BFS is built by grouping multiple servers' memory together. Using the ZooKeeper library, BFS provides a unified consistent file system view over the main memory of commodity servers. BFS uses backend storage to persistently store data and provide availability. BFS does not replicate data and depends on the backend storage if replication is required. BFS provides strong consistency as long as there is no crash in the system. After a crash, BFS consistency guarantees depend on the backend storage. BFS uses the FUSE library to provide a POSIX-like interface. The term POSIX-like is used instead of POSIX-compliant because when BFS is used with a non-consistent backend storage, BFS cannot provide strong consistency guarantees of the POSIX standard. However, using a consistent backend storage ensures BFS a POSIX-compliant system. Finally, BFS has a pluggable networking sub-system that supports different transport modes such as TCP and a zero copying networking mode using the PF_RING library. Utilizing the PF_RING library to bypass the regular network stack, allows BFS to achieve low overhead communication among servers.

Several experiments are designed to understand if a simple design such as BFS is able to fulfill the storage requirements of diskless nodes while delivering a performance comparable to existing more complex solutions. In order to put results of BFS evaluations in perspective, GlusterFS is used for reference. Throughput evaluations of BFS shows that BFS performs similar to in-memory file systems when the dataset fits in the main memory of a server and when remote operations are involved BFS delivers a superior/comparable performance to GlusterFS. In addition, the reliability evaluations of BFS shows that BFS is very efficient in recovering from failures and is only limited by the backend storage.

Furthermore, SPEC SFS 2014 is used to measure the scalability and the latency of different file operations. The SPEC benchmark results strongly indicate that using ZooKeeper to build the namespace limits BFS in handling a large number of files. Moreover, SPEC evaluations show that BFS is a highly efficient design when applications and BFS servers are co-located because BFS reads are done at the memory speed, while GlusterFS can not utilize memory due to consistency. Finally, evaluations of the use of PF_RING for communication (BFS_ZERO) among servers versus TCP indicate that BFS_ZERO significantly reduces network latency compared to BFS_TCP.

In conclusion, BFS is a highly simple design which provides a superior/comparable performance compared to other more complex solutions. BFS simple design indicates that not all the complexity of existing solutions such as GlusterFS is required to satisfy storage requirements of a diskless environment. However, certain areas in BFS such as handling large number of files need to be improved upon. BFS is still at an early development stage, and many revealed shortcomings as well as other improvements are considered to be studied in future releases.

APPENDICES

Appendix A

ZooKeeper Architecture

ZooKeeper is a service for coordinating processes of distributed applications using a shared hierarchical namespace. ZooKeeper provides many of distributed applications' needs, such as synchronization, failure/membership detection, leader election, and group messaging in a replicated centralized service. The ZooKeeper service is replicated over a set of machines to achieve high availability and performance. A client can connect to any of these machines, and ZooKeeper guarantees a single unified image of service regardless of the server the client connects to. Moreover, ZooKeeper guarantees atomicity and sequential consistency of updates issued by a client; meaning all updates either succeed or fail, and they are applied in the order they are issued by the client. In order to achieve these guarantees for updates, ZooKeeper uses a leader based atomic broadcast protocol [25]. In addition, ZooKeeper uses a quorum based schema for updates (majority consensus), and all updates pass through the leader. However, because a typical workload of ZooKeeper is dominated by read operations rather than updates, ZooKeeper serves read requests locally from each server. Figure A.1 depicts an overview of the ZooKeeper architecture; it can be seen that a client can connect to any of ZooKeeper ensemble servers, and each server can serve clients read requests locally, whereas all write request should go through the leader node.

Serving read requests locally by each server can lead to strong consistency models. For instance, *Monotonic Reads* model states if a client reads a from data item x, any subsequent read from x should return a or a more recent version of a. Although eventually all servers in a ZooKeeper ensemble will have the latest version of a data item, there might be short temporal inconsistency in the result of read operations. This behaviour might not be desired in every application, and ZooKeeper provides a *SYNC* read facility that guarantees to return the latest version of each data item at the cost of a longer read delay.

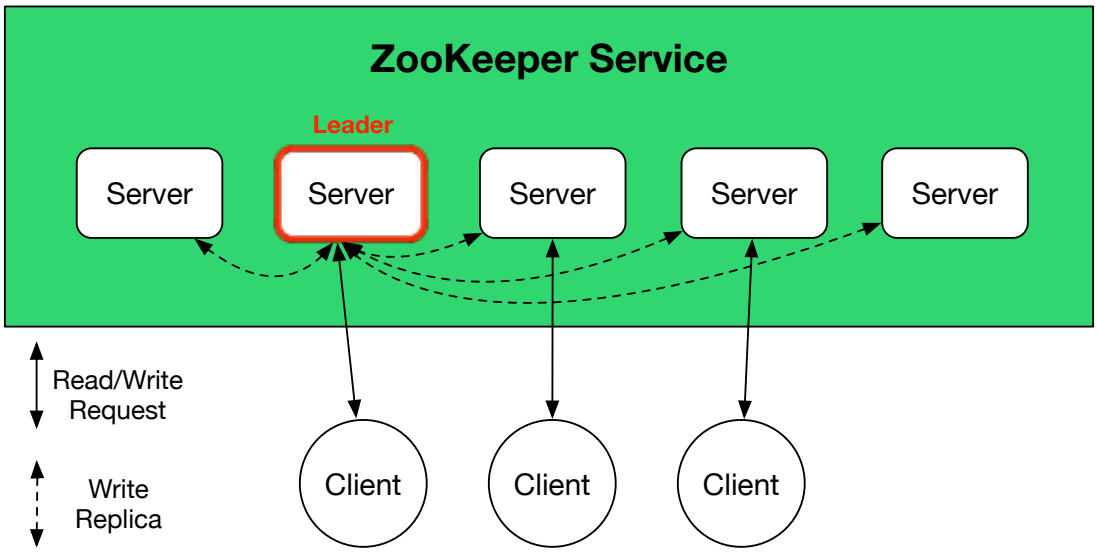


Figure A.1: ZooKeeper Architecture

Appendix B

Testbed Configuration

- 1x Head node: Supermicro SSG-6047R-E1R36L
  - 2x Intel E5-2630v2 CPU
  - 256 GB RAM
  - 14x 2TB 7200RPM SAS2 hard drives (LSI HBA-connected)
  - 1x Intel S3700 400 GB SATA3 SSD
  - 4x Intel i350 gigabit Ethernet ports
  - 1x Mellanox 40 GbE QSFP port
- 15x Computer nodes: Supermicro SYS-6017R-TDF
  - 2x Intel E5-2620v2 CPU
  - 64 GB RAM
  - 3x 1TB SATA3 hard drives
  - 1x Intel S3700 200 GB SATA3 SSD
  - 2x Intel i350 gigabit Ethernet ports
  - 1x Mellanox 10 GbE SFP port
- All nodes run Debian GNU/Linux 8.0 (jessie) with Linux kernel 3.16.0-4-amd64

Cluster nodes are connected to each other using a Mellanox SX1012 10/40 GbE and a SSE-G2252 48-port Gigabit switch.

Appendix C

SPEC SFS 2014 File Operations

Operation	Description
read()	Read file data sequentially
read_file()	Read an entire file sequentially
read_random()	Read file data at random offsets in the files
write()	Write file data sequentially
write_file()	Write an entire file sequentially
write_random()	Write file data at random offsets in the files
rmw()	Read+modify+write file data at random offsets in files
mkdir()	Create a directory
unlink()	Unlink/remove a file
append()	Append to the end of an existing file
lock()	Lock a file
unlock()	Unlock a file
access()	Perform the access() system call on a file
stat()	Perform the stat() system call on a file
chmod()	Perform the chmod() system call on a file
create()	Create a new file
readdir()	Perform a readdir() system call on a directory
statfs()	Perform the statfs() system call on a file system
copyfile()	Copy a file

Table C.1: SPEC SFS 2014 File Operations, Taken directly from [18]

Appendix D

SPEC SFS 2014 Configuration File

Content of configuration file (sfs_rc):

```
BENCHMARK={SWBUILD|DATABASE|VDA|VDI}
LOAD=2
INCR_LOAD=1
NUM_RUNS=20
CLIENT_MOUNTPOINTS=localhost:/home/behrooz/git/BFS/mountdir
EXEC_PATH=/opt/specsfs/binaries/linux/x64/netmist
USER=behrooz
WARMUP_TIME=300
IPV6_ENABLE=0
PRIME_MON_SCRIPT=
PRIME_MON_ARGS=
NETMIST_LOGS=
```

References

[1] Amazon S3. http://aws.amazon.com/s3/. Accessed: 2015-07-22.

[2] CDMI. http://www.snia.org/cdmi. Accessed: 2015-07-22.

[3] FS-Cache. https://www.kernel.org/doc/Documentation/filesystems/caching/fscache.txt. Accessed: 2015-07-22.

[4] fsync-tester. https://github.com/gregsfortytwo/fsync-tester. Accessed: 2015-07-22.

[5] FUSE. http://fuse.sourceforge.net. Accessed: 2015-07-22.

[6] GlusterFS. http://www.gluster.org. Accessed: 2015-07-22.

[7] Google Cloud Storage. https://cloud.google.com/storage/. Accessed: 2015-07-22.

[8] Hadoop. https://hadoop.apache.org. Accessed: 2015-07-22.

[9] IEEE 802.1Qbb Standard. https://standards.ieee.org/findstds/standard/802.1Qbb-2011.html. Accessed: 2015-07-22.

[10] ioping. https://code.google.com/p/ioping/. Accessed: 2015-07-22.

[11] Iozone. http://www.iozone.org. Accessed: 2015-07-22.

[12] KOS. https://git.uwaterloo.ca/mkarsten/KOS. Accessed: 2015-08-13.

[13] Lustre. http://lustre.org/documentation/. Accessed: 2015-07-22.

[14] Openstack Swift. http://docs.openstack.org/developer/swift/. Accessed: 2015-07-22.

[15] PF_RING. http://www.ntop.org/products/packet-capture/pf_ring/. Accessed: 2015-07-22.

[16] POSIX. http://pubs.opengroup.org/onlinepubs/9699919799/. Accessed: 2015-07-22.

[17] SGI OmniStor. https://www.sgi.com/products/storage/. Accessed: 2015-07-22.

[18] SPEC SFS 2014. https://www.spec.org/sfs2014/. Accessed: 2015-07-22.

[19] yaffs. http://www.yaffs.net. Accessed: 2015-07-22.

[20] Joe Arnold. *OpenStack Swift. Using, Administering, and Developing for Swift Object Storage*. O'Reilly Media, 1005 Gravenstein Highway North Sebastopol, CA 95472. USA, 2014.

[21] Eric A. Brewer. Towards Robust Distributed Systems. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[22] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[23] In Hwan Doh, Young Jin Kim, Jung Soo Park, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Towards Greener Data Centers with Storage Class Memory: Minimizing Idle Power Waste Through Coarse-grain Management in Fine-grain Scale. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, pages 309–318, New York, NY, USA, 2010. ACM.

[24] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[25] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[26] John H Howard. An Overview of the Andrew File System. In *Winter 1988 USENIX Conference Proceedings*, pages 23–26, 1988.

[27] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[28] Jaegeuk Kim, Hyotaek Shim, Seon-Yeong Park, Seungryoul Maeng, and Jin-Soo Kim. FlashLight: A Lightweight Flash File System for Embedded Systems. *ACM Trans. Embed. Comput. Syst.*, 11S(1):18:1–18:23, June 2012.

[29] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[30] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 273–286, Berkeley, CA, USA, 2015. USENIX Association.

[31] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 6:1–6:15, New York, NY, USA, 2014. ACM.

[32] Michael Kuhn. memfs A FUSE Memory File System. *Ruprecht-Karls-Universit at Heidelberg*, 2008.

[33] Jim Mostek, Bill Earl, Steven Levine, Steve Lord, Russell Cattelan, Ken McDonell, Ted Kline, Brian Gaffey, and Rajagopal Ananthanarayanan. Porting the SGI XFS File System to Linux. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00, pages 34–34, Berkeley, CA, USA, 2000. USENIX Association.

[34] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A Read/Write Peer-to-peer File System. *SIGOPS Oper. Syst. Rev.*, 36(SI):31–44, December 2002.

[35] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar,

Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMCloud. *Commun. ACM*, 54(7):121–130, July 2011.

[36] Swapnil V. Patil, Garth A. Gibson, Sam Lang, and Milo Polte. GIGA+: Scalable Directories for Shared File Systems. In *Proceedings of the 2Nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, PDSW '07, pages 26–29, New York, NY, USA, 2007. ACM.

[37] K. Salah, R. Al-Shaikh, and M. Sindi. Towards Green Computing Using Diskless High Performance Clusters. In *Proceedings of the 7th International Conference on Network and Services Management*, CNSM '11, pages 456–459, Laxenburg, Austria, Austria, 2011. International Federation for Information Processing.

[38] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. Innovations in Internetworking. chapter Design and Implementation of the Sun Network Filesystem, pages 379–390. Artech House, Inc., Norwood, MA, USA, 1988.

[39] Peter Snyder. tmpfs: A virtual memory file system. In *Proceedings of the Autumn 1990 European UNIX Users Group Conference*, pages 241–248, 1990.

[40] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[41] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[42] Vasily Tarasov, Saumitra Bhanage, Erez Zadok, and Margo Seltzer. Benchmarking File System Benchmarking: It *IS* Rocket Science. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[43] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P. Wright. A Nine Year Study of File System and Storage Benchmarking. *Trans. Storage*, 4(2):5:1–5:56, May 2008.

[44] Theodore Y. Ts'o and Stephen Tweedie. Planned Extensions to the Linux Ext2/Ext3 Filesystem. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 235–243, Berkeley, CA, USA, 2002. USENIX Association.

[45] Feng Wang, Scott A. Brandt, Ethan L. Miller, and Darrell D. E. Long. OBFS: A File System for Object-based Storage Devices. In *Proceedings of th 21st IEEE / 12th NASA GODDARD Conference on Mass Storage Systems and Technologies, College Park, MD*, pages 283–300, 2004.

[46] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[47] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.